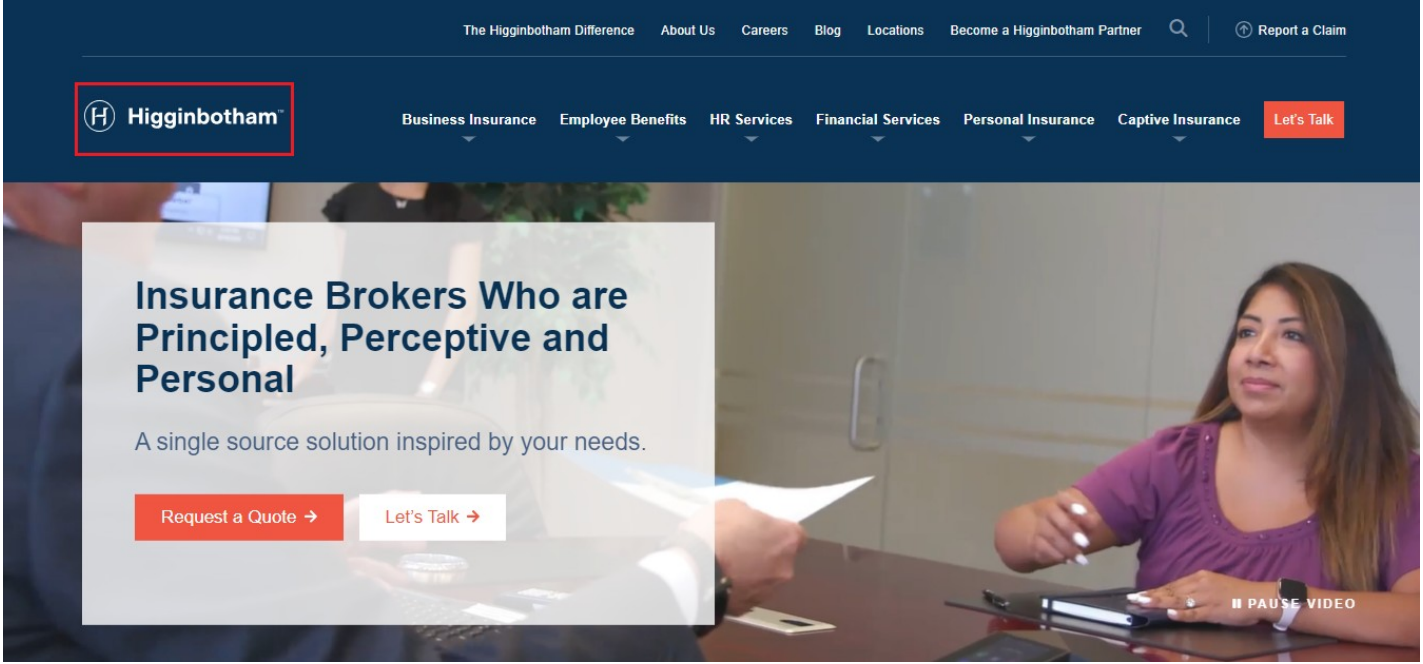


Exhibit 2

<p>US7203844B1</p> <p>1. A method for a recursive security protocol for protecting digital content, comprising:</p>	<p>Higginbotham Companies website higginbotham.com (“The accused instrumentality”)</p> <p>The accused instrumentality practices a method for a recursive security protocol (e.g., TLS 1.3 security protocol) for protecting digital content (e.g., digital certificate related to the accused instrumentality).</p> <p>The accused instrumentality utilizes TLS 1.3 security protocol (hereinafter “the standard”) for communicating content such as digital certificate, application data, etc., with a client. The standard provides a two-level encryption security. It encrypts a plaintext with a first encryption technique and generates a ciphertext. Further, it encrypts the ciphertext with a second encryption technique i.e., recursive encryption security.</p>  <p>https://www.higginbotham.com/</p>
---	---

Google Play


higginbotham

Apps & games Movies & TV Books


Higginbotham FSA

Higginbotham Wex Health Mobile

Save time and hassles with the Higginbotham FSA mobile app



1K+ Downloads

 Everyone

Install

View your **account(s)** and link to resources from **I Want To**

Check your **account activity** anytime, anywhere

Manage **expenses** from a consolidated dashboard

My Accounts

FSA 2020 Plan	\$2,499.00
HRA 2020 Plan	\$1,000.00
ExpenditureCard 2020 Plan	\$2,300.00

I Want To

- File A Claim
- Search for Eligibility
- Manage Expenses
- View and Upload Receipts

Account

FSA 2020 Plan

Details

Available Balance	\$2,499.00
Plan Service Date	10/01/2020
Plan Ending Date	09/30/2021

Account Activity

Payment Deduction	\$15.00	Balance	\$2,484.00
Payment Deduction	\$15.00	Balance	\$2,469.00
Payment Deduction	\$15.00	Balance	\$2,454.00
Payment Deduction	\$15.00	Balance	\$2,439.00
Payment Deduction	\$15.00	Balance	\$2,424.00
Payment Deduction	\$15.00	Balance	\$2,409.00
Payment Deduction	\$15.00	Balance	\$2,394.00

Dashboard

manage expenses

Expenses

Prescription	\$10.00	Unpaid	Pending
Dental	\$10.00	Unpaid	Pending
Medical	\$1.00	Unpaid	Pending
Medical	\$10.00	Unpaid	Pending

<https://play.google.com/store/search?q=higginbotham&c=apps&hl=en&gl=US>

Privacy - Terms



HigginbothamTM

Welcome

Monday, September 2, 2024

You are here: Welcome

Account Login

User name:

Login

[Retrieve login information](#)

<https://higginbotham.secureclient.net/Welcome/tabid/366042/Default.aspx?returnurl=%2f>

Security overview



This page is secure (valid HTTPS).



Certificate - **valid and trusted**

The connection to this site is using a valid, trusted server certificate issued by GlobalSign Atlas R3 DV TLS CA 2024 Q3.

[View certificate](#)



Connection - **secure connection settings**

The connection to this site is encrypted and authenticated using TLS 1.3, X25519, and AES_128_GCM.



Resources - **all served securely**

All resources on this page are served securely.

<https://higginbotham.secureclient.net/Welcome/tabid/366042/Default.aspx?returnurl=%2f>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

As shown below, the accused instrumentality utilizes a two-level algorithm security. It utilizes the SHA256RSA encryption algorithm as a first encryption algorithm i.e., signature encryption algorithm and the TLS_AES_256_GCM_SHA384 encryption algorithm as a second encryption algorithm i.e., AEAD encryption algorithm.

[illegible]

Source: Fiddler Capture

SignedCertTimestamp(RFC6962) empty

ALPN h2, http/1.1

signature alqs ecdsa secp256r1 sha256,rsa_pss_rsae sha256,rsa_pkcs1 sha256,ecdsa secp384r1 sha384,rsa_pss_rsae sha384,rsa_pkcs1 sha384,rsa_pss_rsae sha512,rsa_pkcs1 sha512

0x001b 02 00 02

supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]

key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85

7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7

EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 C8 B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C C8 D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F

C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B 8C 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7

23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 F7 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 00 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 80 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0

11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12

83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 6E 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1

08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB

CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E

2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66

31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 8C 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08

9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C

48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55

CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 8C AD B8 60 BF F2 BC 1E 43 24 23 C8 57 48 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96

B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90

9B AB 18 53 2C E9 0F 05 83 D3 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4

7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24

extended_master_secret empty

ec_point_formats uncompressed [0x0]

status_request OCSP - Implicit Responder

psk_key_exchange_modes 01 01

renegotiation_info 00

0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15

F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E

2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C

E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A

server_name www.higginbotham.com

grease (0x3a3a) 00

Source: Fiddler Capture

SignedCertTimestamp(RFC6962) empty

ALPN h2, http/1.1

signature alqs ecdsa secp256r1 sha256,rsa_pss_rsae sha256,rsa_pkcs1 sha256,ecdsa secp384r1 sha384,rsa_pss_rsae sha384,rsa_pkcs1 sha384,rsa_pss_rsae sha512,rsa_pkcs1 sha512

0x001b 02 00 02

supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]

key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85

7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7

EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 C8 B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C C8 D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F

C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B 8C 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7

23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 F7 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 00 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 80 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0

11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12

83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 6E 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1

08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB

CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E

2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66

31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 8C 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08

9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C

48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55

CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 8C AD B8 60 BF F2 BC 1E 43 24 23 C8 57 48 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96

B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90

9B AB 18 53 2C E9 0F 05 83 D3 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4

7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24

extended_master_secret empty

ec_point_formats uncompressed [0x0]

status_request OCSP - Implicit Responder

psk_key_exchange_modes 01 01

renegotiation_info 00

0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15

F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E

2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C

E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A

server_name www.higginbotham.com

grease (0x3a3a) 00

Source: Fiddler Capture

[Thumbprint]
129157F7A5852AD23DF17FE2934E6396DB6B0251

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption
algorithm

[Public Key]

Algorithm: RSA
Length: 2048

Key Blob: 30 82 01 0a 02 82 01 01 00 ca b8 66 5f 8a 37 94 14 e1 2f e3 49 cf d7 ad cb 20 8c 54 93 6b af 66 56 02 11 26 db 20 13 00 a0 ce fe ae e9 ac 8d 68 8f a0 56 fd ff b4 b1 4b 9c 54 ff 55 4a 95 76 b9 db 3c 77 5e 9a b1 dd 30 74 36 77 cb 92 12 df 17 62 6d 2a aa 74 3c f6 68 c8 34 8d eb 69 97 9e f2 43 02 fa d7 76 99 c0 bc b3 56 2d 17 ca 39 c8 00 5d 91 c8 1d 9b c9 70 c8 c1 b0 da 40 23 3e 9a fe a9 ed e4 60 ce 15 b0 af 43 a3 07 a5 81 25 dd c3 92 e7 72 9e 27 df bd 01 05 2a b4 60 b8 83 6b fa bc db 97 51 47 a8 2e d0 52 7f c2 0a 1c 02 96 d2 c7 fd 04 a5 7e 78 04 ee 59 ed 33 9a e4 cc 42 bb 87 36 95 ac 4b 9b 6b 4f 96 4c ed 20 f7 bd 71 4b bc d8 91 81 3c be 56 c8 05 a6 e1 7f 66 85 88 46 91 96 a3 e4 e8 bb 45 f5 b8 78 25 4b 02 cb 23 e6 54 94 ed 94 f6 87 3d 29 55 31 eb 59 e9 a1 54 4c 50 b0 74 2c a7 b6 1e 7f 23 02 03 01 00 01

Parameters: 05 00

[Extensions]

* Key Usage(2.5.29.15):

Digital Signature, Key Encipherment (a0)

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm
00000019	63 6F 6D 3A 34 34 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 77									com:443 HTTP/1.1..Host: w
00000032	77 77 2E 68 69 67 67 69 6E 62 6F 74 68 61 6D 2E 63 6F 6D 3A 34 34 33 0D 0A									ww.migginbotham.com:443..
0000004B	43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 6B 65 65 70 2D 61 6C 69 76 65 0D 0A 55									Connection: keep-alive..U
00000064	73 65 72 2D 41 67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 57									ser-Agent: Mozilla/5.0 (W
0000007D	69 6E 64 6F 77 73 20 4E 54 20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36									indows NT10.0; Win64; x6
00000096	34 29 20 41 70 70 6C 65 57 65 62 4B 69 74 2F 35 33 37 2E 33 36 20 28 4B 48									4) AppleWebKit/537.36 (KH
000000AF	54 4D 4C 2C 20 6C 69 6B 65 20 47 65 63 6B 6F 29 20 43 68 72 6F 6D 65 2F 31									TML, like Gecko) Chrome/1
000000C8	32 36 2E 30 2E 30 2E 30 20 53 61 66 61 72 69 2F 35 33 37 2E 33 36 0D 0A 0D									26.0.0.0 Safari/537.36...
000000E1	0A 41 20 53 53 4C 76 33 2D 63 6F 6D 70 61 74 69 62 6C 65 20 43 6C 69 65 6E									.A SSLv3-compatible Clie
000000FA	74 48 65 6C 6C 6F 20 68 61 6E 64 73 68 61 6B 65 20 77 61 73 20 66 6F 75 6E									tHello handshake was foun
00000113	64 2E 20 46 69 64 64 6C 65 72 20 65 78 74 72 61 63 74 65 64 20 74 68 65 20									d. Fiddler extracted the
0000012C	70 61 72 61 6D 65 74 65 72 73 20 62 65 6C 6F 77 2E 0A 0A 53 65 63 75 72 65									parameters below...Secure
00000145	20 50 72 6F 74 6F 63 6F 6C 3A 20 54 4C 53 20 31 2E 33 0A 43 69 70 68 65 72									Protocol: TLS 1.3.Cipher
0000015E	20 53 75 69 74 65 3A 20 54 4C 53 5F 41 45 53 5F 32 35 36 5F 47 43 4D 5F 53									Suite: TLS_AES_256_GCM_S
00000177	48 41 33 38 34 0A 0A 52 65 63 6F 72 64 20 4C 61 79 65 72 20 56 65 72 73 69									HA384..Record Layer Versi
00000190	6F 6E 3A 20 33 2E 33 20 28 54 4C 53 2F 31 2E 32 29 0A 52 61 6E 64 6F 6D 3A									on: 3.3 (TLS/1.2).Random:
000001A9	20 46 36 20 39 44 20 31 45 20 30 35 20 33 44 20 35 38 20 35 33 20 35 30 20									F6 9D 1E 05 3D 58 53 50
000001C2	43 35 20 33 38 20 43 42 20 36 38 20 45 39 20 42 31 20 37 31 20 42 45 20 30									C5 38 CB 68 E9 B1 71 BE 0
000001DB	32 20 37 37 20 41 37 20 46 41 20 41 42 20 33 46 20 43 43 20 31 44 20 39 37									2 77 A7 FA AB 3F CC 1D 97
000001F4	20 31 42 20 34 43 20 41 46 20 41 42 20 37 41 20 43 44 20 36 39 0A 22 54 69									1B 4C AF AB 7A CD 69."Ti
0000020D	6D 65 22 3A 20 32 31 2D 30 39 2D 31 39 37 32 20 30 38 3A 33 33 3A 31 38 0A									me": 21-09-1972 08:33:18.
00000226	53 65 73 73 69 6F 6E 49 44 3A 20 35 45 20 42 33 20 34 42 20 37 30 20 31 32									SessionID: 5E B3 4B 70 12
0000023F	20 34 44 20 32 43 20 43 42 20 36 41 20 35 42 20 39 41 20 36 33 20 38 39 20									4D 2C CB 6A 5B 9A 63 89

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream																								
000004C9	36	37	20	39	38	20	39	46	20	38	42	20	38	38	20	44	39	20	41	30	20	43	36	20	31	67	98	9F	8B	88	D9	A0	C6	1
000004E2	35	20	43	43	20	33	37	20	42	42	20	35	31	20	39	31	20	31	44	20	33	30	20	43	45	5	CC	37	BB	51	91	1D	30	CE
000004FB	20	38	44	20	46	36	20	38	45	20	35	46	20	35	34	20	43	32	20	35	41	20	45	33	20	8D	F6	8E	5F	54	C2	5A	E3	
00000514	38	39	20	32	43	20	37	42	20	38	39	20	41	41	20	42	42	20	43	35	20	41	46	20	45	39	2C	7B	89	AA	BB	C5	AF	E
0000052D	39	20	35	36	20	31	30	20	42	38	20	31	35	20	35	37	20	39	34	20	41	31	20	41	31	9	56	10	B8	15	57	94	A1	A1
00000546	20	30	43	20	32	41	20	37	33	20	33	45	20	42	34	20	38	35	20	37	41	20	39	35	20	0C	2A	73	3E	B4	85	7A	95	
0000055F	34	34	20	33	34	20	30	37	20	31	39	20	43	38	20	33	33	20	33	46	20	45	43	20	39	44	34	07	19	C8	33	3F	EC	9
00000578	43	20	34	35	20	34	36	20	30	42	20	38	34	20	43	39	20	31	43	20	32	32	20	45	31	C	45	46	0B	84	C9	1C	22	E1
00000591	20	33	43	20	39	46	20	38	37	20	37	31	20	41	33	20	35	42	20	43	33	20	37	32	20	3C	9F	87	71	A3	5B	C3	72	
000005AA	42	30	20	45	31	20	43	39	20	44	35	20	43	41	20	39	44	20	43	34	20	36	39	20	42	B0	E1	C9	D5	CA	9D	C4	69	B
000005C3	30	20	30	35	20	35	42	20	39	44	20	45	32	20	41	34	20	32	42	20	35	36	20	46	35	0	05	5B	9D	E2	A4	2B	56	F5
000005DC	20	42	46	20	38	32	20	38	34	20	36	37	20	39	43	20	43	36	20	36	35	20	35	46	20	BF	82	84	67	9C	C6	65	5F	
000005F5	38	32	20	33	45	20	42	32	20	46	30	20	42	42	20	46	35	20	33	42	20	41	46	20	30	82	3E	B2	F0	BB	F5	3B	AF	0
0000060E	46	20	38	36	20	31	31	20	41	34	20	31	43	20	30	30	20	35	41	20	34	36	20	36	39	F	86	11	A4	1C	00	5A	46	69
00000627	20	44	39	20	35	37	20	41	46	20	33	31	20	43	36	20	43	32	20	46	34	20	46	30	20	D9	57	AF	31	C6	C2	F4	F0	
00000640	33	38	20	46	37	20	45	43	20	39	37	20	36	41	20	32	31	20	37	38	20	43	34	20	41	38	F7	EC	97	6A	21	78	C4	A
00000659	38	20	41	41	20	42	34	20	38	35	20	43	45	20	43	39	20	38	35	20	36	37	20	44	37	8	AA	B4	85	CE	C9	85	67	D7
00000672	20	37	41	20	30	43	20	45	42	20	41	42	20	37	39	20	31	44	20	38	33	20	33	41	20	7A	0C	EB	AB	79	1D	83	3A	
0000068B	42	32	20	33	42	20	36	30	20	44	36	20	41	33	20	43	32	20	30	31	20	38	37	20	41	B2	3B	60	D6	A3	C2	01	87	A
000006A4	37	20	46	37	20	43	43	20	42	38	20	41	30	20	45	35	20	31	45	20	31	46	20	35	35	7	F7	CC	B8	A0	E5	1E	1F	55
000006BD	20	38	38	20	33	34	20	36	35	20	32	46	20	42	41	20	43	41	20	30	36	20	39	43	20	88	34	65	2F	BA	CA	06	9C	
000006D6	33	42	20	31	37	20	39	44	20	41	36	20	31	31	20	42	32	20	33	39	20	39	46	20	33	3B	17	9D	A6	11	B2	39	9F	3
000006EF	34	20	30	30	20	36	43	20	43	42	20	44	30	20	35	33	20	33	45	20	35	37	20	31	36	4	00	6C	CB	D0	53	3E	57	16

Source: Fiddler Capture

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3

Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate ==

[Version]

V3

[Subject]

CN=higginbotham.com

Simple Name: higginbotham.com

DNS Name: higginbotham.com

[Issuer]

CN=GTS CA 1P5, O=Google Trust Services LLC, C=US

Second decryption
algorithm

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type.

The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

Second
encryption

- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.

First
encryption

- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.
- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Introduction

The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties:

- Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated.

Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK).

- Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques.
- Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection.

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2¹⁴ bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding `TLSPlaintext.length` due to the inclusion of `TLSInnerPlaintext.type` and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.




<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (`iv_length`) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [\[RFC5116\], Section 4](#)). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>This specification defines the following cipher suites for use with TLS 1.3.</p> <table border="1"> <thead> <tr> <th>Description</th><th>Value</th></tr> </thead> <tbody> <tr> <td>TLS_AES_128_GCM_SHA256</td><td>{0x13,0x01}</td></tr> <tr> <td>TLS_AES_256_GCM_SHA384</td><td>{0x13,0x02}</td></tr> <tr> <td>TLS_CHACHA20_POLY1305_SHA256</td><td>{0x13,0x03}</td></tr> <tr> <td>TLS_AES_128_CCM_SHA256</td><td>{0x13,0x04}</td></tr> <tr> <td>TLS_AES_128_CCM_8_SHA256</td><td>{0x13,0x05}</td></tr> </tbody> </table> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>	Description	Value	TLS_AES_128_GCM_SHA256	{0x13,0x01}	TLS_AES_256_GCM_SHA384	{0x13,0x02}	TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}	TLS_AES_128_CCM_SHA256	{0x13,0x04}	TLS_AES_128_CCM_8_SHA256	{0x13,0x05}
Description	Value												
TLS_AES_128_GCM_SHA256	{0x13,0x01}												
TLS_AES_256_GCM_SHA384	{0x13,0x02}												
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}												
TLS_AES_128_CCM_SHA256	{0x13,0x04}												
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}												
encrypting a bitstream with a first encryption algorithm;	<p>The standard practices encrypting a bitstream (e.g., bitstream of digital certificate) with a first encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA encryption algorithm).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext.</p>												

Security overview



This page is secure (valid HTTPS).

Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

View certificate

Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

As shown below, the accused instrumentality discloses the signature encryption algorithm.

The screenshot displays a Fiddler capture window. On the left, a list of network sessions is shown, with session 8 highlighted as a tunnel to www.higginbotham.com:443. The main pane on the right shows the details of this tunnel, including the 'Secure Protocol: TLS 1.3' and 'Cipher Suite: TLS_AES_256_GCM_SHA384', which are highlighted with a red box. Below this, the server certificate details are visible, including the subject 'CN=higginbotham.com' and the issuer 'CN=GTS CA 1P5, O=Google Trust Services LLC, C=US'.

Source: Fiddler Capture

	<div><div>SignedCertTimestamp(RFC6962) empty</div><div>ALPN h2, http/1.1</div><div>signature alqs ecdsa secp256r1 sha256,rsa_pss_rsae sha256,rsa_pkcs1 sha256,ecdsa secp384r1 sha384,rsa_pss_rsae sha384,rsa_pkcs1 sha384,rsa_pss_rsae sha512,rsa_pkcs1 sha512</div><div>0x001b 02 00 02</div><div>supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]</div><div>key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85</div><div>7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7</div><div>EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 06 E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F</div><div>C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B 8C 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 5B BD DC 3C 1E 5D 8A 7E 1A B6 2E F7</div><div>23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 B7 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0</div><div>11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 89 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12</div><div>83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1</div><div>08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB</div><div>CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E</div><div>2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66</div><div>31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 8C 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08</div><div>9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C</div><div>48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55</div><div>CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 8C AD B8 60 BF F2 BC 1E 43 34 23 C8 57 48 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96</div><div>B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90</div><div>9B AB 18 53 2C E9 0F 05 83 D3 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4</div><div>7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24</div><div>extended_master_secret empty</div><div>ec_point_formats uncompressed [0x0]</div><div>status_request OCSP - Implicit Responder</div><div>psk_key_exchange_modes 01 01</div><div>renegotiation_info 00</div><div>0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15</div><div>F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E</div><div>2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C</div><div>E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A</div><div>server_name www.higginbotham.com</div><div>grease (0x3a3a) 00</div></div>
Source: Fiddler Capture	
	<div><div>SignedCertTimestamp(RFC6962) empty</div><div>ALPN h2, http/1.1</div><div>signature alqs ecdsa secp256r1 sha256,rsa_pss_rsae sha256,rsa_pkcs1 sha256,ecdsa secp384r1 sha384,rsa_pss_rsae sha384,rsa_pkcs1 sha384,rsa_pss_rsae sha512,rsa_pkcs1 sha512</div><div>0x001b 02 00 02</div><div>supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]</div><div>key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85</div><div>7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7</div><div>EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 06 E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F</div><div>C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B 8C 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 5B BD DC 3C 1E 5D 8A 7E 1A B6 2E F7</div><div>23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 B7 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0</div><div>11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 89 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12</div><div>83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1</div><div>08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB</div><div>CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E</div><div>2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66</div><div>31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 8C 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08</div><div>9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C</div><div>48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55</div><div>CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 8C AD B8 60 BF F2 BC 1E 43 34 23 C8 57 48 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96</div><div>B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90</div><div>9B AB 18 53 2C E9 0F 05 83 D3 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4</div><div>7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24</div><div>extended_master_secret empty</div><div>ec_point_formats uncompressed [0x0]</div><div>status_request OCSP - Implicit Responder</div><div>psk_key_exchange_modes 01 01</div><div>renegotiation_info 00</div><div>0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15</div><div>F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E</div><div>2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C</div><div>E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A</div><div>server_name www.higginbotham.com</div><div>grease (0x3a3a) 00</div></div>
Source: Fiddler Capture	

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<-----> [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.
- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.
- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A First encryption "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.
- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```




<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>Introduction</p> <p>The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties:</p> <ul style="list-style-type: none"> - Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK). - Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques. - Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection. <p>https://datatracker.ietf.org/doc/html/rfc8446</p>
<p>associating a first decryption algorithm with the encrypted bit stream;</p>	<p>The standard practices associating a first decryption algorithm (e.g., signature decryption algorithm i.e., SHA256RSA decryption algorithm) with the encrypted bit stream (e.g., encrypted certificate with signature encryption algorithm).</p> <p>The standard practices providing a two-level encryption security for data</p>

communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext.

The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate.

Security overview



This page is secure (valid HTTPS).

- Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

View certificate
- Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.
- Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

As shown below, the accused instrumentality discloses the signature decryption algorithm.

[Thumbprint]
129157F7A5852AD23DF17FE2934E6396DB6B0251

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption
algorithm

[Public Key]

Algorithm: RSA
Length: 2048

Key Blob: 30 82 01 0a 02 82 01 01 00 ca b8 66 5f 8a 37 94 14 e1 2f e3 49 cf d7 ad cb 20 8c 54 93 6b af 66 56 02 11 26 db 20 13 00 a0 ce fe ae e9 ac 8d 68 8f a0 56 fd ff b4 b1 4b 9c 54 ff 55 4a 95 76 b9 db 3c 77 5e 9a b1 dd 30 74 36 77 cb 92 12 df 17 62 6d 2a aa 74 3c f6 68 c8 34 8d eb 69 97 9e f2 43 02 fa d7 76 99 c0 bc b3 56 2d 17 ca 39 c8 00 5d 91 c8 1d 9b c9 70 c8 c1 b0 da 40 23 3e 9a fe a9 ed e4 60 ce 15 b0 af 43 a3 07 a5 81 25 dd c3 92 e7 72 9e 27 df bd 01 05 2a b4 60 b8 83 6b fa bc db 97 51 47 a8 2e d0 52 7f c2 0a 1c 02 96 d2 c7 fd 04 a5 7e 78 04 ee 59 ed 33 9a e4 cc 42 bb 87 36 95 ac 4b 9b 6b 4f 96 4c ed 20 f7 bd 71 4b bc d8 91 81 3c be 56 c8 05 a6 e1 7f 66 85 88 46 91 96 a3 e4 e8 bb 45 f5 b8 78 25 4b 02 cb 23 e6 54 94 ed 94 f6 87 3d 29 55 31 eb 59 e9 a1 54 4c 50 b0 74 2c a7 b6 1e 7f 23 02 03 01 00 01

Parameters: 05 00

[Extensions]

* Key Usage(2.5.29.15):
Digital Signature, Key Encipherment (a0)

Source: Fiddler Capture

OID description

First decryption algorithm identifier

OID:	{iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-1(1) sha256WithRSAEncryption(11)}	(ASN.1 notation)
	1.2.840.113549.1.1.11	(dot notation)
	/ISO/Member-Body/US/113549/1/1/11	(OID-IRI notation)

Description:

Public-Key Cryptography Standards (PKCS) #1 version 1.5 signature algorithm with Secure Hash Algorithm 256 (SHA256) and Rivest, Shamir and Adleman (RSA) encryption

<http://oid-info.com/get/1.2.840.113549.1.1.11>

-- When the following OIDs are used in an AlgorithmIdentifier, the
-- parameters MUST be present and MUST be NULL.

sha224WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 14 }

sha256WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 11 }

sha384WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 12 }

sha512WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 13 }

<https://www.ietf.org/rfc/rfc4055.txt>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<-----> [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<----- [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.

- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A First encryption "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.

- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

First
decryption
algorithm
information

The algorithm field specifies the signature algorithm used (see [Section 4.2.3](#) for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in [Section 4.4.1](#), namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. we know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$




First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

	<p>The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A <i>cryptographic hash function</i> is a function that computes a <i>message authentication code</i> from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.</p> <p>https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf</p>
<p>encrypting both the encrypted bit stream and the first decryption algorithm with a second encryption algorithm to yield a second bit stream;</p>	<p>The standard practices encrypting both the encrypted bit stream (e.g., encrypted digital certificate) and the first decryption algorithm (e.g., signature decryption algorithm) with a second encryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) to yield a second bit stream (e.g., TLS ciphertext bitstream).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</p> <p>The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it.</p>

The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.

Security overview

This page is secure (valid HTTPS).

- Certificate - valid and trusted
The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.
[View certificate](#)
- Connection - secure connection settings
The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.
- Resources - all served securely
All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm
00000019	63 6F 6D 3A 34 34 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 77									con:443 HTTP/1.1..Host: w
00000032	77 77 2E 68 69 67 67 69 6E 62 6F 74 68 61 6D 2E 63 6F 6D 3A 34 34 33 0D 0A									ww.higginbotham.com:443..
0000004B	43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 6B 65 65 70 2D 61 6C 69 76 65 0D 0A 55									Connection: keep-alive..U
00000064	73 65 72 2D 41 67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 57									ser-Agent: Mozilla/5.0 (W
0000007D	69 6E 64 6F 77 73 20 4E 54 20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36									indows NT 10.0; Win64; x6
00000096	34 29 20 41 70 70 6C 65 57 65 62 4B 69 74 2F 35 33 37 2E 33 36 20 28 4B 48									4) AppleWebKit/537.36 (KH
000000AF	54 4D 4C 2C 20 6C 69 6B 65 20 47 65 63 6B 6F 29 20 43 68 72 6F 6D 65 2F 31									TML, like Gecko) Chrome/1
000000C8	32 36 2E 30 2E 30 2E 30 20 53 61 66 61 72 69 2F 35 33 37 2E 33 36 0D 0A 0D									26.0.0.0 Safari/537.36...
000000E1	0A 41 20 53 53 4C 76 33 2D 63 6F 6D 70 61 74 69 62 6C 65 20 43 6C 69 65 6E									.A SSLv3-compatible Clie
000000FA	74 48 65 6C 6C 6F 20 68 61 6E 64 73 68 61 6B 65 20 77 61 73 20 66 6F 75 6E									tHello handshake was foun
00000113	64 2E 20 46 69 64 64 6C 65 72 20 65 78 74 72 61 63 74 65 64 20 74 68 65 20									d. Fiddler extracted the
0000012C	70 61 72 61 6D 65 74 65 72 73 20 62 65 6C 6F 77 2E 0A 0A 53 65 63 75 72 65									parameters below...Secure
00000145	20 50 72 6F 74 6F 63 6F 6C 3A 20 54 4C 53 20 31 2E 33 0A 43 69 70 68 65 72									Protocol: TLS 1.3.Cipher
0000015E	20 53 75 69 74 65 3A 20 54 4C 53 5F 41 45 53 5F 32 35 36 5F 47 43 4D 5F 53									Suite: TLS_AES_256_GCM_S
00000177	48 41 33 38 34 0A 0A 52 65 63 6F 72 64 20 4C 61 79 65 72 20 56 65 72 73 69									HA384..Record Layer Versi
00000190	6F 6E 3A 20 33 2E 33 20 28 54 4C 53 2F 31 2E 32 29 0A 52 61 6E 64 6F 6D 3A									on: 3.3 (TLS/1.2).Random:
000001A9	20 46 36 20 39 44 20 31 45 20 30 35 20 33 44 20 35 38 20 35 33 20 35 30 20									F6 9D 1E 05 3D 58 53 50
000001C2	43 35 20 33 38 20 43 42 20 36 38 20 45 39 20 42 31 20 37 31 20 42 45 20 30									C5 38 CB 68 E9 B1 71 BE 0
000001DB	32 20 37 37 20 41 37 20 46 41 20 41 42 20 33 46 20 43 43 20 31 44 20 39 37									2 77 A7 FA AB 3F CC 1D 97
000001F4	20 31 42 20 34 43 20 41 46 20 41 42 20 37 41 20 43 44 20 36 39 0A 22 54 69									1B 4C AF AB 7A CD 69."Ti
0000020D	6D 65 22 3A 20 32 31 2D 30 39 2D 31 39 37 32 20 30 38 3A 33 33 3A 31 38 0A									me": 21-09-1972 08:33:18.
00000226	53 65 73 73 69 6F 6E 49 44 3A 20 35 45 20 42 33 20 34 42 20 37 30 20 31 32									SessionID: 5E B3 4B 70 12
0000023F	20 34 44 20 32 43 20 43 42 20 36 41 20 35 42 20 39 41 20 36 33 20 38 39 20									4D 2C CB 6A 5B 9A 63 89

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream
000004C9	36 37 20 39 38 20 39 46 20 38 42 20 38 38 20 44 39 20 41 30 20 43 36 20 31									67 98 9F 8B 88 D9 A0 C6 1
000004E2	35 20 43 43 20 33 37 20 42 42 20 35 31 20 39 31 20 31 44 20 33 30 20 43 45									5 CC 37 BB 51 91 1D 30 CE
000004FB	20 38 44 20 46 36 20 38 45 20 35 46 20 35 34 20 43 32 20 35 41 20 45 33 20									8D F6 8E 5F 54 C2 5A E3
00000514	38 39 20 32 43 20 37 42 20 38 39 20 41 41 20 42 42 20 43 35 20 41 46 20 45									39 2C 7B 89 AA BB C5 AF E
0000052D	39 20 35 36 20 31 30 20 42 38 20 31 35 20 35 37 20 39 34 20 41 31 20 41 31									9 56 10 B8 15 57 94 A1 A1
00000546	20 30 43 20 32 41 20 37 33 20 33 45 20 42 34 20 38 35 20 37 41 20 39 35 20									0C 2A 73 3E B4 85 7A 95
0000055F	34 34 20 33 34 20 30 37 20 31 39 20 43 38 20 33 33 20 33 46 20 45 43 20 39									44 34 07 19 C8 33 3F EC 9
00000578	43 20 34 35 20 34 36 20 30 42 20 38 34 20 43 39 20 31 43 20 32 32 20 45 31									C 45 46 0B 84 C9 1C 22 E1
00000591	20 33 43 20 39 46 20 38 37 20 37 31 20 41 33 20 35 42 20 43 33 20 37 32 20									3C 9F 87 71 A3 5B C3 72
000005AA	42 30 20 45 31 20 43 39 20 44 35 20 43 41 20 39 44 20 43 34 20 36 39 20 42									B0 E1 C9 D5 CA 9D C4 69 B
000005C3	30 20 30 35 20 35 42 20 39 44 20 45 32 20 41 34 20 32 42 20 35 36 20 46 35									0 05 5B 9D E2 A4 2B 56 F5
000005DC	20 42 46 20 38 32 20 38 34 20 36 37 20 39 43 20 43 36 20 36 35 20 35 46 20									BF 82 84 67 9C C6 65 5F
000005F5	38 32 20 33 45 20 42 32 20 46 30 20 42 42 20 46 35 20 33 42 20 41 46 20 30									82 3E B2 F0 BB F5 3B AF 0
0000060E	46 20 38 36 20 31 31 20 41 34 20 30 31 43 20 30 30 20 35 41 20 34 36 20 36 39									F 86 11 A4 1C 00 5A 46 69
00000627	20 44 39 20 35 37 20 41 46 20 33 31 20 43 36 20 43 32 20 46 34 20 46 30 20									D9 57 AF 31 C6 C2 F4 F0
00000640	33 38 20 46 37 20 45 43 20 39 37 20 36 41 20 32 31 20 37 38 20 43 34 20 41									38 F7 EC 97 6A 21 78 C4 A
00000659	38 20 41 41 20 42 34 20 38 35 20 43 45 20 43 39 20 38 35 20 36 37 20 44 37									8 AA B4 85 CE C9 85 67 D7
00000672	20 37 41 20 30 43 20 45 42 20 41 42 20 37 39 20 31 44 20 38 33 20 33 41 20									7A 0C EB AB 79 1D 83 3A
0000068B	42 32 20 33 42 20 36 30 20 44 36 20 41 33 20 43 32 20 30 31 20 38 37 20 41									B2 3B 60 D6 A3 C2 01 87 A
000006A4	37 20 46 37 20 43 43 20 42 38 20 41 30 20 45 35 20 31 45 20 31 46 20 35 35									7 F7 CC B8 A0 E5 1E 1F 55
000006BD	20 38 38 20 33 34 20 36 35 20 32 46 20 42 41 20 43 41 20 30 36 20 39 43 20									88 34 65 2F BA CA 06 9C
000006D6	33 42 20 31 37 20 39 44 20 41 36 20 31 31 20 42 32 20 33 39 20 39 46 20 33									3B 17 9D A6 11 B2 39 9F 3
000006EF	34 20 30 30 20 36 43 20 43 42 20 44 30 20 35 33 20 33 45 20 35 37 20 31 36									4 00 6C CB D0 53 3E 57 16

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext

handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2¹⁴ bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding `TLSPlaintext.length` due to the inclusion of `TLSInnerPlaintext.type` and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (`iv_length`) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [\[RFC5116\], Section 4](#)). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

All handshake messages after the ServerHello are now encrypted.
The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<----- [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

Second
encryption

- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.

First
encryption

- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.

- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

First
decryption
algorithm
information

The algorithm field specifies the signature algorithm used (see [Section 4.2.3](#) for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in [Section 4.4.1](#), namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>The <u>"extension_data" field of these extensions contains a SignatureSchemeList value:</u></p> <pre> enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), </pre> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>associating a second decryption algorithm with the second bit stream.</p>	<p>The standard practices associating a second decryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) with the second bit stream (e.g., TLS ciphertext bitstream).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</p>

The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

					Transformer	Headers	TextView	SyntaxView	ImageView	HexView	WebView	Auth	Caching	Cookies	Raw	JSON	XML
8	200	HTTP	Tunnel to www.higginbotham.com:443	3,612	Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.												
9	200	HTTPS	www.higginbotham.com /	17,353	Secure Protocol: TLS 1.3 Cipher Suite: TLS_AES_256_GCM_SHA384												
10	200	HTTPS	www.googleapis.com /oauth2/v4/token	399	== Server Certificate == [Version] V3												
11	200	HTTPS	optimizationguide-p... /v1:GetHints?	428	[Subject] CN=higginbotham.com Simple Name: higginbotham.com DNS Name: higginbotham.com												
12	200	HTTP	Tunnel to safebrowsing.google.com...	10,052	[Issuer] CN=GTS CA 1P5, O=Google Trust Services LLC, C=US Simple Name: GTS CA 1P5 DNS Name: GTS CA 1P5												
13	200	HTTPS	safebrowsing.google.com /safebrowsing/clientrepor...	32													
14	200	HTTPS	www.higginbotham.com /wp-includes/css/dist/bloc...	14,716													
15	200	HTTPS	www.higginbotham.com /wp-content/cache/min/1/...	35,238													
16	200	HTTP	Tunnel to cdnjs.cloudflare.com:443	2,748													
17	200	HTTP	Tunnel to cdn.cloudflare.com:443	4,277													
18	200	HTTPS	www.higginbotham.com /wp-content/cache/min/1/...	6,980													
19	200	HTTPS	www.higginbotham.com /wp-content/cache/min/1/...	23,260													
20	200	HTTPS	www.higginbotham.com /wp-content/cache/min/1/...	23,266													
21	200	HTTPS	www.higginbotham.com /wp-content/themes/orbit...	26,907													
22	200	HTTPS	www.higginbotham.com /wp-content/cache/min/1/...	1,436													
23	200	HTTPS	www.higginbotham.com /wp-includes/js/jquery/jqu...	31,396													

Source: Fiddler Capture

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3

Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate =====
[Version]
V3

Second decryption
algorithm

[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com

[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream
000004C9	36 37 20 39 38 20 39 46 20 38 42 20 38 38 20 44 39 20 41 30 20 43 36 20 31									67 98 9F 8B 88 D9 A0 C6 1
000004E2	35 20 43 43 20 33 37 20 42 42 20 35 31 20 39 31 20 31 44 20 33 30 20 43 45									5 CC 37 BB 51 91 1D 30 CE
000004FB	20 38 44 20 46 36 20 38 45 20 35 46 20 35 34 20 43 32 20 35 41 20 45 33 20									8D F6 8E 5F 54 C2 5A E3
00000514	38 39 20 32 43 20 37 42 20 38 39 20 41 41 20 42 42 20 43 35 20 41 46 20 45									39 2C 7B 89 AA BB C5 AF E
0000052D	39 20 35 36 20 31 30 20 42 38 20 31 35 20 35 37 20 39 34 20 41 31 20 41 31									9 56 10 B8 15 57 94 A1 A1
00000546	20 30 43 20 32 41 20 37 33 20 33 45 20 42 34 20 38 35 20 37 41 20 39 35 20									0C 2A 73 3E B4 85 7A 95
0000055F	34 34 20 33 34 20 30 37 20 31 39 20 43 38 20 33 33 20 33 46 20 45 43 20 39									44 34 07 19 C8 33 3F EC 9
00000578	43 20 34 35 20 34 36 20 30 42 20 38 34 20 43 39 20 31 43 20 32 32 20 45 31									C 45 46 0B 84 C9 1C 22 E1
00000591	20 33 43 20 39 46 20 38 37 20 37 31 20 41 33 20 35 42 20 43 33 20 37 32 20									3C 9F 87 71 A3 5B C3 72
000005AA	42 30 20 45 31 20 43 39 20 44 35 20 43 41 20 39 44 20 43 34 20 36 39 20 42									B0 E1 C9 D5 CA 9D C4 69 B
000005C3	30 20 30 35 20 35 42 20 39 44 20 45 32 20 41 34 20 32 42 20 35 36 20 46 35									0 05 5B 9D E2 A4 2B 56 F5
000005DC	20 42 46 20 38 32 20 38 34 20 36 37 20 39 43 20 43 36 20 36 35 20 35 46 20									BF 82 84 67 9C C6 65 5F
000005F5	38 32 20 33 45 20 42 32 20 46 30 20 42 42 20 46 35 20 33 42 20 41 46 20 30									82 3E B2 F0 BB F5 3B AF 0
0000060E	46 20 38 36 20 31 31 20 41 34 20 31 43 20 30 30 20 35 41 20 34 36 20 36 39									F 86 11 A4 1C 00 5A 46 69
00000627	20 44 39 20 35 37 20 41 46 20 33 31 20 43 36 20 43 32 20 46 34 20 46 30 20									D9 57 AF 31 C6 C2 F4 F0
00000640	33 38 20 46 37 20 45 43 20 39 37 20 36 41 20 32 31 20 37 38 20 43 34 20 41									38 F7 EC 97 6A 21 78 C4 A
00000659	38 20 41 41 20 42 34 20 38 35 20 43 45 20 43 39 20 38 35 20 36 37 20 44 37									8 AA B4 85 CE C9 85 67 D7
00000672	20 37 41 20 30 43 20 45 42 20 41 42 20 37 39 20 31 44 20 38 33 20 33 41 20									7A 0C EB AB 79 1D 83 3A
0000068B	42 32 20 33 42 20 36 30 20 44 36 20 41 33 20 43 32 20 30 31 20 38 37 20 41									B2 3B 60 D6 A3 C2 01 87 A
000006A4	37 20 46 37 20 43 43 20 42 38 20 41 30 20 45 35 20 31 45 20 31 46 20 35 35									7 F7 CC B8 A0 E5 1E 1F 55
000006BD	20 38 38 20 33 34 20 36 35 20 32 46 20 42 41 20 43 41 20 30 36 20 39 43 20									88 34 65 2F BA CA 06 9C
000006D6	33 42 20 31 37 20 39 44 20 41 36 20 31 31 20 42 32 20 33 39 20 39 46 20 33									3B 17 9D A6 11 B2 39 9F 3
000006EF	34 20 30 30 20 36 43 20 43 42 20 44 30 20 35 33 20 33 45 20 35 37 20 31 36									4 00 6C CB D0 53 3E 57 16

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS

communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2¹⁴ bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.2. Authenticated Decryption

Second decryption algorithm

The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block.

These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<----- [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>The <u>"extension_data" field of these extensions contains a SignatureSchemeList value:</u></p> <pre> enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), </pre> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>2. The method of claim 1, further comprising decrypting the first bit stream and the second</p>	<p>The standard further discloses decrypting the first bit stream (e.g., encrypted digital certificate with signature encryption algorithm i.e., SHA-256 RSA, etc.) and the second bit stream (e.g., a second-level encryption with AEAD encryption algorithm such as TLS AES 256 GCM SHA384, etc.) with the first associated decryption</p>

<p>bit stream with the first associated decryption algorithm and the second associated decryption algorithm wherein the decryption is accomplished by a target unit.</p>	<p>algorithm (e.g., signature decryption algorithm i.e., SHA-256 RSA, etc.) and the second associated decryption algorithm (e.g., cipher suit selected from one of the AEAD decryption algorithms such as TLS_AES_256_GCM_SHA384, etc.) wherein the decryption is accomplished by a target unit (e.g., a server of the accused instrumentality).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</p> <p>The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.</p>
--	---

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

Transformer					Headers	TextView	SyntaxView	ImageView	HexView	WebView	Auth	Caching	Cookies	Raw	JSON	XML
Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.																
Secure Protocol: TLS 1.3																
Cipher Suite: TLS_AES_256_GCM_SHA384																
== Server Certificate ==																
[Version]																
V3																
[Subject]																
CN=higginbotham.com																
Simple Name: higginbotham.com																
DNS Name: higginbotham.com																
[Issuer]																
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US																
Simple Name: GTS CA 1P5																
DNS Name: GTS CA 1P5																

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 C8 B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B BC 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 05 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 BC 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 BC AD B8 60 BF F2 BC 1E 43 34 23 C8 57 48 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 C8 B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B BC 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 05 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 BC 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 BC AD B8 60 BF F2 BC 1E 43 34 23 C8 57 48 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

[Thumbprint]
129157F7A5852AD23DF17FE2934E6396DB6B0251

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption
algorithm

[Public Key]

Algorithm: RSA
Length: 2048

Key Blob: 30 82 01 0a 02 82 01 01 00 ca b8 66 5f 8a 37 94 14 e1 2f e3 49 cf d7 ad cb 20 8c 54 93 6b af 66 56 02 11 26 db 20 13 00 a0 ce fe ae e9 ac 8d 68 8f a0 56 fd ff b4 b1 4b 9c 54 ff 55 4a 95 76 b9 db 3c 77 5e 9a b1 dd 30 74 36 77 cb 92 12 df 17 62 6d 2a aa 74 3c f6 68 c8 34 8d eb 69 97 9e f2 43 02 fa d7 76 99 c0 bc b3 56 2d 17 ca 39 c8 00 5d 91 c8 1d 9b c9 70 c8 c1 b0 da 40 23 3e 9a fe a9 ed e4 60 ce 15 b0 af 43 a3 07 a5 81 25 dd c3 92 e7 72 9e 27 df bd 01 05 2a b4 60 b8 83 6b fa bc db 97 51 47 a8 2e d0 52 7f c2 0a 1c 02 96 d2 c7 fd 04 a5 7e 78 04 ee 59 ed 33 9a e4 cc 42 bb 87 36 95 ac 4b 9b 6b 4f 96 4c ed 20 f7 bd 71 4b bc d8 91 81 3c be 56 c8 05 a6 e1 7f 66 85 88 46 91 96 a3 e4 e8 bb 45 f5 b8 78 25 4b 02 cb 23 e6 54 94 ed 94 f6 87 3d 29 55 31 eb 59 e9 a1 54 4c 50 b0 74 2c a7 b6 1e 7f 23 02 03 01 00 01

Parameters: 05 00

[Extensions]

* Key Usage(2.5.29.15):
Digital Signature, Key Encipherment (a0)

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm
00000019	63 6F 6D 3A 34 34 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 77									com:443 HTTP/1.1..Host: w
00000032	77 77 2E 68 69 67 67 69 6E 62 6F 74 68 61 6D 2E 63 6F 6D 3A 34 34 33 0D 0A									ww.migginbotham.com:443..
0000004B	43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 6B 65 65 70 2D 61 6C 69 76 65 0D 0A 55									Connection: keep-alive..U
00000064	73 65 72 2D 41 67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 57									ser-Agent: Mozilla/5.0 (W
0000007D	69 6E 64 6F 77 73 20 4E 54 20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36									indows NT10.0; Win64; x6
00000096	34 29 20 41 70 70 6C 65 57 65 62 4B 69 74 2F 35 33 37 2E 33 36 20 28 4B 48									4) AppleWebKit/537.36 (KH
000000AF	54 4D 4C 2C 20 6C 69 6B 65 20 47 65 63 6B 6F 29 20 43 68 72 6F 6D 65 2F 31									TML, like Gecko) Chrome/1
000000C8	32 36 2E 30 2E 30 2E 30 20 53 61 66 61 72 69 2F 35 33 37 2E 33 36 0D 0A 0D									26.0.0.0 Safari/537.36...
000000E1	0A 41 20 53 53 4C 76 33 2D 63 6F 6D 70 61 74 69 62 6C 65 20 43 6C 69 65 6E									.A SSLv3-compatible Clie
000000FA	74 48 65 6C 6C 6F 20 68 61 6E 64 73 68 61 6B 65 20 77 61 73 20 66 6F 75 6E									tHello handshake was foun
00000113	64 2E 20 46 69 64 64 6C 65 72 20 65 78 74 72 61 63 74 65 64 20 74 68 65 20									d. Fiddler extracted the
0000012C	70 61 72 61 6D 65 74 65 72 73 20 62 65 6C 6F 77 2E 0A 0A 53 65 63 75 72 65									parameters below...Secure
00000145	20 50 72 6F 74 6F 63 6F 6C 3A 20 54 4C 53 20 31 2E 33 0A 43 69 70 68 65 72									Protocol: TLS 1.3.Cipher
0000015E	20 53 75 69 74 65 3A 20 54 4C 53 5F 41 45 53 5F 32 35 36 5F 47 43 4D 5F 53									Suite: TLS_AES_256_GCM_S
00000177	48 41 33 38 34 0A 0A 52 65 63 6F 72 64 20 4C 61 79 65 72 20 56 65 72 73 69									HA384..Record Layer Versi
00000190	6F 6E 3A 20 33 2E 33 20 28 54 4C 53 2F 31 2E 32 29 0A 52 61 6E 64 6F 6D 3A									on: 3.3 (TLS/1.2).Random:
000001A9	20 46 36 20 39 44 20 31 45 20 30 35 20 33 44 20 35 38 20 35 33 20 35 30 20									F6 9D 1E 05 3D 58 53 50
000001C2	43 35 20 33 38 20 43 42 20 36 38 20 45 39 20 42 31 20 37 31 20 42 45 20 30									C5 38 CB 68 E9 B1 71 BE 0
000001DB	32 20 37 37 20 41 37 20 46 41 20 41 42 20 33 46 20 43 43 20 31 44 20 39 37									2 77 A7 FA AB 3F CC 1D 97
000001F4	20 31 42 20 34 43 20 41 46 20 41 42 20 37 41 20 43 44 20 36 39 0A 22 54 69									1B 4C AF AB 7A CD 69."Ti
0000020D	6D 65 22 3A 20 32 31 2D 30 39 2D 31 39 37 32 20 30 38 3A 33 33 3A 31 38 0A									me": 21-09-1972 08:33:18.
00000226	53 65 73 73 69 6F 6E 49 44 3A 20 35 45 20 42 33 20 34 42 20 37 30 20 31 32									SessionID: 5E B3 4B 70 12
0000023F	20 34 44 20 32 43 20 43 42 20 36 41 20 35 42 20 39 41 20 36 33 20 38 39 20									4D 2C CB 6A 5B 9A 63 89

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream																								
000004C9	36	37	20	39	38	20	39	46	20	38	42	20	38	38	20	44	39	20	41	30	20	43	36	20	31	67	98	9F	8B	88	D9	A0	C6	1
000004E2	35	20	43	43	20	33	37	20	42	42	20	35	31	20	39	31	20	31	44	20	33	30	20	43	45	5	CC	37	BB	51	91	1D	30	CE
000004FB	20	38	44	20	46	36	20	38	45	20	35	46	20	35	34	20	43	32	20	35	41	20	45	33	20	8D	F6	8E	5F	54	C2	5A	E3	
00000514	38	39	20	32	43	20	37	42	20	38	39	20	41	41	20	42	42	20	43	35	20	41	46	20	45	39	2C	7B	89	AA	BB	C5	AF	E
0000052D	39	20	35	36	20	31	30	20	42	38	20	31	35	20	35	37	20	39	34	20	41	31	20	41	31	9	56	10	B8	15	57	94	A1	A1
00000546	20	30	43	20	32	41	20	37	33	20	33	45	20	42	34	20	38	35	20	37	41	20	39	35	20	0C	2A	73	3E	B4	85	7A	95	
0000055F	34	34	20	33	34	20	30	37	20	31	39	20	43	38	20	33	33	20	33	46	20	45	43	20	39	44	34	07	19	C8	33	3F	EC	9
00000578	43	20	34	35	20	34	36	20	30	42	20	38	34	20	43	39	20	31	43	20	32	32	20	45	31	C	45	46	0B	84	C9	1C	22	E1
00000591	20	33	43	20	39	46	20	38	37	20	37	31	20	41	33	20	35	42	20	43	33	20	37	32	20	3C	9F	87	71	A3	5B	C3	72	
000005AA	42	30	20	45	31	20	43	39	20	44	35	20	43	41	20	39	44	20	43	34	20	36	39	20	42	B0	E1	C9	D5	CA	9D	C4	69	B
000005C3	30	20	30	35	20	35	42	20	39	44	20	45	32	20	41	34	20	32	42	20	35	36	20	46	35	0	05	5B	9D	E2	A4	2B	56	F5
000005DC	20	42	46	20	38	32	20	38	34	20	36	37	20	39	43	20	43	36	20	36	35	20	35	46	20	BF	82	84	67	9C	C6	65	5F	
000005F5	38	32	20	33	45	20	42	32	20	46	30	20	42	42	20	46	35	20	33	42	20	41	46	20	30	82	3E	B2	F0	BB	F5	3B	AF	0
0000060E	46	20	38	36	20	31	31	20	41	34	20	31	43	20	30	30	20	35	41	20	34	36	20	36	39	F	86	11	A4	1C	00	5A	46	69
00000627	20	44	39	20	35	37	20	41	46	20	33	31	20	43	36	20	43	32	20	46	34	20	46	30	20	D9	57	AF	31	C6	C2	F4	F0	
00000640	33	38	20	46	37	20	45	43	20	39	37	20	36	41	20	32	31	20	37	38	20	43	34	20	41	38	F7	EC	97	6A	21	78	C4	A
00000659	38	20	41	41	20	42	34	20	38	35	20	43	45	20	43	39	20	38	35	20	36	37	20	44	37	8	AA	B4	85	CE	C9	85	67	D7
00000672	20	37	41	20	30	43	20	45	42	20	41	42	20	37	39	20	31	44	20	38	33	20	33	41	20	7A	0C	EB	AB	79	1D	83	3A	
0000068B	42	32	20	33	42	20	36	30	20	44	36	20	41	33	20	43	32	20	30	31	20	38	37	20	41	B2	3B	60	D6	A3	C2	01	87	A
000006A4	37	20	46	37	20	43	43	20	42	38	20	41	30	20	45	35	20	31	45	20	31	46	20	35	35	7	F7	CC	B8	A0	E5	1E	1F	55
000006BD	20	38	38	20	33	34	20	36	35	20	32	46	20	42	41	20	43	41	20	30	36	20	39	43	20	88	34	65	2F	BA	CA	06	9C	
000006D6	33	42	20	31	37	20	39	44	20	41	36	20	31	31	20	42	32	20	33	39	20	39	46	20	33	3B	17	9D	A6	11	B2	39	9F	3
000006EF	34	20	30	30	20	36	43	20	43	42	20	44	30	20	35	33	20	33	45	20	35	37	20	31	36	4	00	6C	CB	D0	53	3E	57	16

Source: Fiddler Capture

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3

Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate ==

[Version]
V3

[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com

[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US

Second decryption
algorithm

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type.

The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^{14} bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.2. Authenticated Decryption

Second decryption algorithm

The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block.

These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

Client

Server

Key ^ ClientHello

$$\text{Exch} \mid + \text{key_share}^*$$

```
+ signature_algorithms*
```

```
| + psk_key_exchange_modes*
```

$$v + \text{pre_shared_key}^* \rightarrow$$

ServerHello ^ Key

+ key_share* | Exch

$$+ \text{pre_shared_key}^*$$

`{EncryptedExtensions}` ^ Server

```
{CertificateRequest*} v Params
```

```
{Certificate*}
```

```
{CertificateVerify*} | Auth
```

```
{Finished}
```

```
<----- [Application Data*]
```

Digital Content

$$\wedge \{ \text{Certificate}^* \}$$

Auth	{CertificateVerify*}
------	----------------------

v {Finished}

[Application Data]

----->

```
<-----> [Application Data]
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

	<p>The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A <i>cryptographic hash function</i> is a function that computes a <i>message authentication code</i> from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.</p> <p>https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf</p>
3. The method of claim 2, wherein the decrypting is done using a key associated with each decryption algorithm.	<p>The standard practices the method such that the decrypting is done using a key (e.g., decryption key) associated with each decryption algorithm (e.g., signature decryption algorithm such as SHA-256RSA, etc., and AEAD decryption algorithm such as TLS_AES_256_GCM_SHA384, etc.).</p>

The Higginbotham Difference About Us Careers Blog Locations Become a Higginbotham Partner Search Report a Claim

Higginbotham Business Insurance Employee Benefits HR Services Financial Services Personal Insurance Captive Insurance Let's Talk

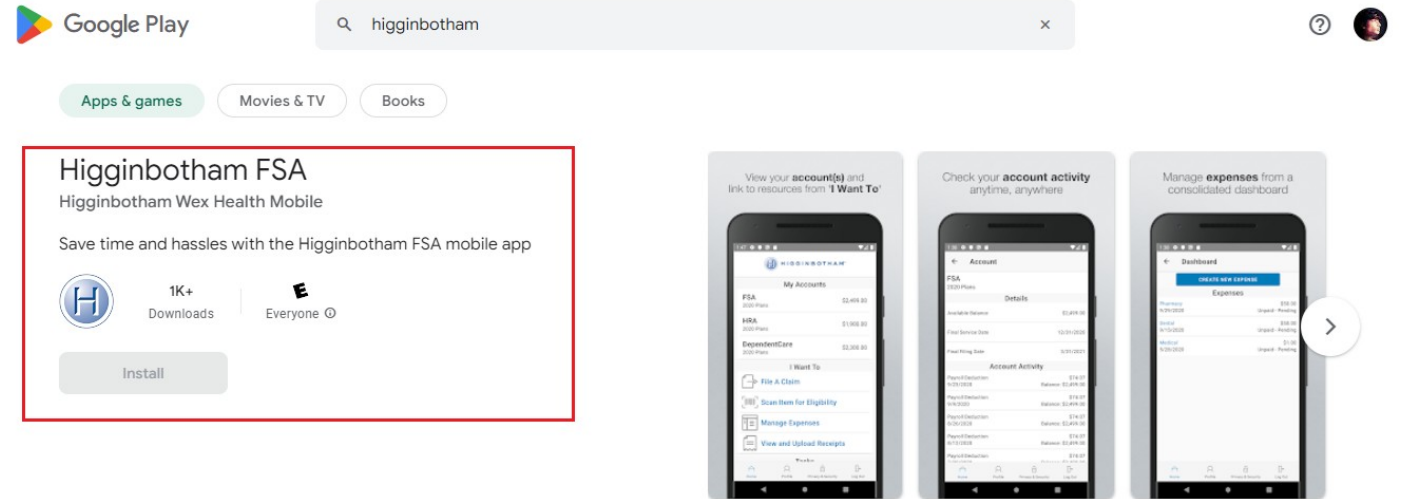
Insurance Brokers Who are Principled, Perceptive and Personal

A single source solution inspired by your needs.

Request a Quote -> Let's Talk ->

PAUSE VIDEO

<https://www.higginbotham.com/>



<https://play.google.com/store/search?q=higginbotham&c=apps&hl=en&gl=US>

```
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
  extended_master_secret  empty
  ec_point_formats  uncompressed [0x0]
  status_request  OCSP - Implicit Responder
  psk_key_exchange_modes  01 01
  renegotiation_info  00
  0xfe0d  00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 DD 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
  server_name  www.higginbotham.com
  csease (0x3a3a)  00
```

Source: Fiddler Capture

As shown below, the signature decryption algorithm utilizes a private key for a first decryption and the AEAD decryption algorithm uses a key K. Both the decryption techniques are decrypting using their respective associated keys.

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2.2. Authenticated Decryption

The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4. The method of claim 3, wherein the key is resident in hardware of the target unit or the key is retrieved from a server.

The standard utilized by the accused instrumentality practices the method such that the key is resident in hardware (e.g., stored in a memory storage of the server such as a database, RAM, etc.) of the target unit (e.g., server of the accused instrumentality) or the key is retrieved from a server.

The Higginbotham Difference About Us Careers Blog Locations Become a Higginbotham Partner Search Report a Claim

Higginbotham Business Insurance Employee Benefits HR Services Financial Services Personal Insurance Captive Insurance Let's Talk

Insurance Brokers Who are Principled, Perceptive and Personal

A single source solution inspired by your needs.

Request a Quote -> Let's Talk ->

PAUSE VIDEO

<https://www.higginbotham.com/>

higginbotham

?

Apps & games

Movies & TV

Books

Higginbotham FSA

Higginbotham Wex Health Mobile

Save time and hassles with the Higginbotham FSA mobile app

1K+

Downloads

E

Everyone

Install

View your **account(s)** and link to resources from **I Want To**

Check your **account activity** anytime, anywhere

Manage **expenses** from a consolidated dashboard

<https://play.google.com/store/search?q=higginbotham&c=apps&hl=en&gl=US>

```

JA 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
    extended_master_secret    empty
    ec_point_formats          uncompressed [0x0]
    status_request            OCSP - Implicit Responder
    psk_key_exchange_modes    01 01
    renegotiation_info        00
                                00 00 01 00 18 00 20 08 F8 C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D0 29 05 15
                                0xfedf
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 88 BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C A8 B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 4A A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
    server_name                www.higginbotham.com
    oresa (0x3a3a)            00

```

Source: Fiddler Capture



Tech Accelerator

Server hardware guide: Architecture, products and management

3. Random access memory



RAM is the main type of memory in a computing system.

RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>



Tech Accelerator

Server hardware guide: Architecture, products and management

f

X

in



4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the [hard disk drive \(HDD\)](#) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

<https://www.techtargget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>

As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

<https://datatracker.ietf.org/doc/html/rfc8446#>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2.2. Authenticated Decryption

The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5. The method of claim 4, wherein the key is contained in a key data structure.

The standard utilized by the accused instrumentality practices the method such that the key (e.g., private key, Key K, etc.) is contained in a key data structure (e.g., data structure).

The Higginbotham Difference About Us Careers Blog Locations Become a Higginbotham Partner Report a Claim

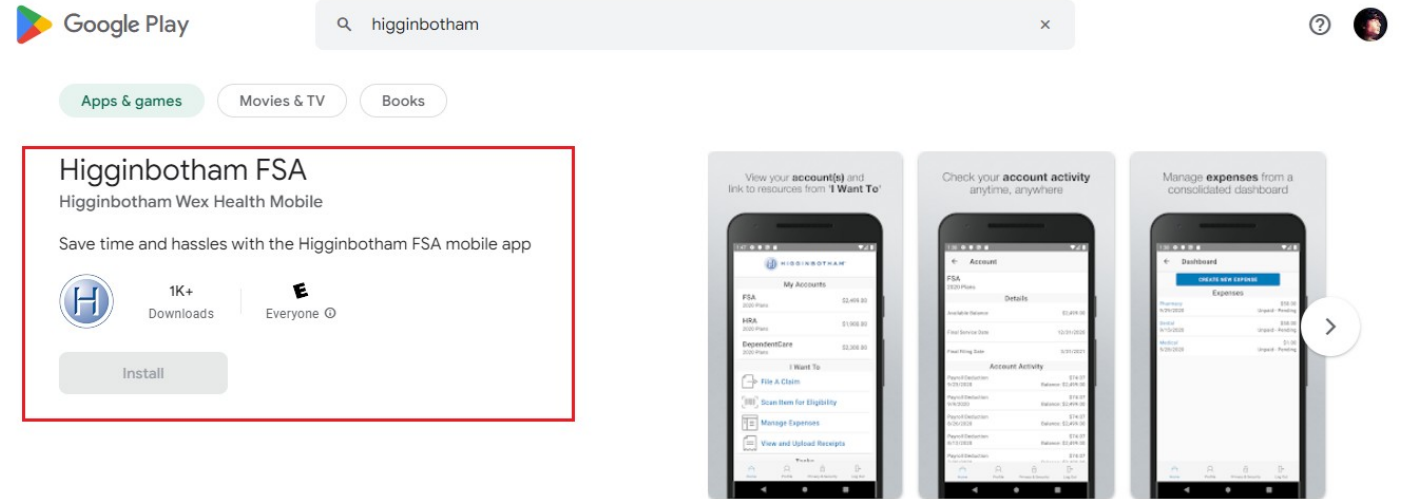
Higginbotham™

Business Insurance Employee Benefits HR Services Financial Services Personal Insurance Captive Insurance

Insurance Brokers Who are Principled, Perceptive and Personal

A single source solution inspired by your needs.

<https://www.higginbotham.com/>



<https://play.google.com/store/search?q=higginbotham&c=apps&hl=en&gl=US>

```
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 DD 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
orease(0x3a3a) 00
```

Source: Fiddler Capture

The accused instrumentality utilizes a server to establish a secure TLS communication with a client. The server must comprise a memory storage and store data according to a data structure to implement the standard efficiently.



Tech Accelerator

Server hardware guide: Architecture, products and management

3. Random access memory



RAM is the main type of memory in a computing system.

RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as

data to be moved to a storage device. Thus, RAM works very

closely with the processor and must match the processor's

incredible speed and performance. This kind of fast memory

is usually termed dynamic RAM, and several DRAM

variations are available for servers.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>



Tech Accelerator

Server hardware guide: Architecture, products and management



4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>

A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it.

In computer science and computer programming, a data structure may be selected or designed to store data for the purpose of using it with various algorithms. In some cases, the algorithm's basic operations are tightly coupled to the data structure's design. Each data structure contains information about the data values, relationships between the data and -- in some cases -- functions that can be applied to the data.

<https://www.techtarget.com/searchdatamanagement/definition/data-structure>

As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

	<p>Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. <u>This is necessary for the ClientHello storage mechanism described in Section 8.2</u> because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.</p> <p>https://datatracker.ietf.org/doc/html/rfc8446#</p>
--	--

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p><u>2.2. Authenticated Decryption</u></p> <p>The <u>authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).</u></p> <p>https://datatracker.ietf.org/doc/html/rfc5116</p> <p>The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. <u>The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.</u></p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>11. The method of claim 3, wherein each encryption algorithm is a symmetric key system or an asymmetric key system.</p>	<p>The standard practices the method such that each encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA, etc., and AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.) is a symmetric key system (e.g., AEAD encryption algorithm, etc.) or an asymmetric key system (e.g., signature encryption algorithm).</p> <p>As shown below, the server comprises a memory storage to store messages for</p>

establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. The standard defines the signature encryption algorithm as an asymmetric cryptography algorithm and the AEAD encryption algorithm as the symmetric cryptography algorithm.

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

<https://datatracker.ietf.org/doc/html/rfc8446#>

Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK).

<https://datatracker.ietf.org/doc/html/rfc8446#section-4>

cipher_suites: A list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. Values are defined in [Appendix B.4](#). If the list contains cipher suites that the server does not recognize, support, or wish to use, the server MUST ignore those cipher suites and process the remaining ones as usual. If the client is attempting a PSK key establishment, it SHOULD advertise at least one cipher suite indicating a Hash associated with the PSK.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

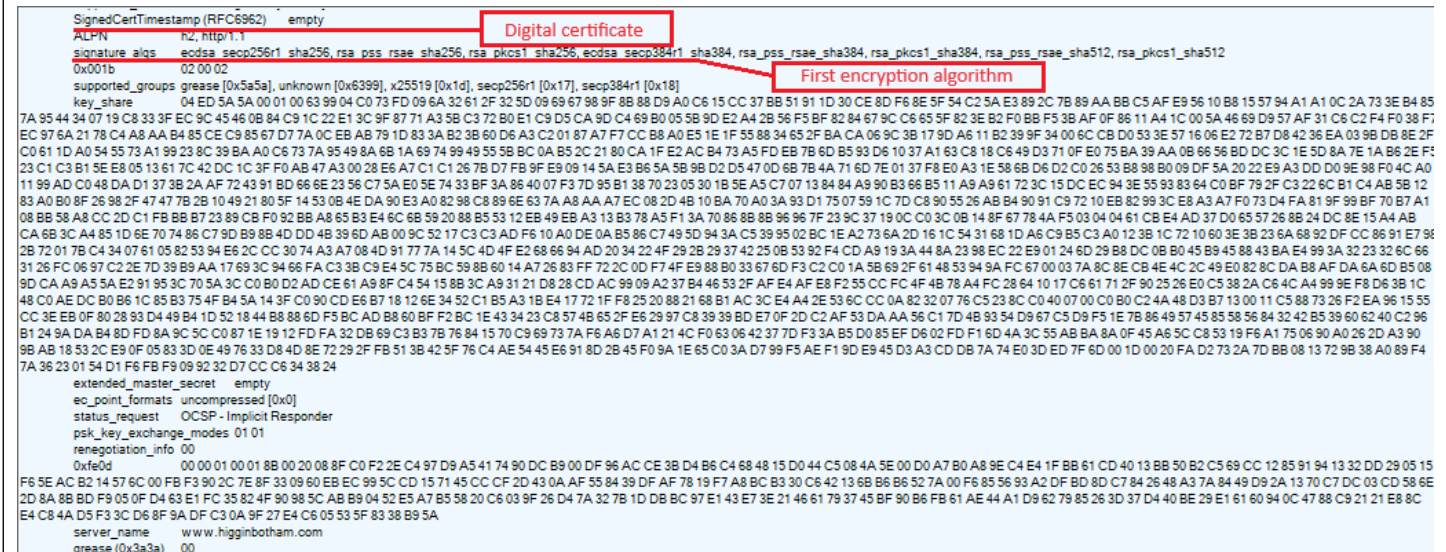
The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

	<p><u>2.2. Authenticated Decryption</u></p> <p>The <u>authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).</u></p> <p>https://datatracker.ietf.org/doc/html/rfc5116</p> <p>The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. <u>The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.</u></p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>12. The method of claim 3, further comprising associating a first Message Authentication Code (MAC) or first digital signature with each</p>	<p>The standard practices associating a first Message Authentication Code (MAC) (e.g., message authentication code with hashing function) or first digital signature with each encrypted bit stream (e.g., encrypted bit stream with the signature encryption algorithm i.e., SHA256RSA, etc., and encrypted bitstream with the AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.).</p> <p>As shown below, the standard discloses a hashing function with each of the encryption</p>

encrypted bit stream.

algorithm. It performs a message authentication code with the utilized hashing function.



```

SignedCertTimestamp(RFC6962) empty
ALPN h2,http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x3a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 1D 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 58 C3 72 B0 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F3 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 06 E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B BC 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F5
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F 0A B7 43 00 28 E6 A7 C1 C1 26 7B D7 FB 9F E9 09 14 5A E3 B6 5A 5B 9B D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD D0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 84 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 6E 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 BC 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 A2 E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 6B 38 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 BC AD B8 60 BF F2 BC 1E 43 34 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 A6 D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 82 C5 69 CC 12 85 91 94 13 32 D0 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
grease (0x3a3a) 00
  
```

Source: Fiddler Capture

First encryption algorithm

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm																									
00000019	63	6F	6D	3A	34	34	33	20	48	54	54	50	2F	31	2E	31	0D	0A	48	6F	73	74	3A	20	77	com:443 HTTP/1.1..Host: w									
00000032	77	77	2E	68	69	67	67	69	6E	62	6F	74	68	61	6D	2E	63	6F	6D	3A	34	34	33	0D	0A	ww.bigginbotham.com:443..									
0000004B	43	6F	6E	6E	65	63	74	69	6F	6E	3A	20	6B	65	65	70	2D	61	6C	69	76	65	0D	0A	55	Connection: keep-alive..U									
00000064	73	65	72	2D	41	67	65	6E	74	3A	20	4D	6F	7A	69	6C	6E	61	2F	35	2E	30	20	28	57	ser-Agent: Mozilla/5.0 (w									
0000007D	69	6E	64	6F	77	73	20	4E	54	20	31	30	2E	30	3B	20	57	69	6E	36	34	3B	20	78	36	indows NT10.0; Win64; x6									
00000096	34	29	20	41	70	70	6C	65	57	65	62	4B	69	74	2F	35	33	37	2E	33	36	20	28	4B	48	4) AppleWebKit/537.36 (KH									
000000AF	54	4D	4C	2C	20	6C	69	6B	65	20	47	65	63	6B	6F	29	20	43	68	72	6F	6D	65	2F	31	TML, like Gecko) Chrome/1									
000000C8	32	36	2E	30	2E	30	2E	30	20	53	61	66	61	72	69	2F	35	33	37	2E	33	36	0D	0A	0D	26.0.0.0 Safari/537.36...									
000000E1	0A	41	20	53	53	4C	76	33	2D	63	6F	6D	70	61	74	69	62	6C	65	20	43	6C	69	65	6E	.A SSLv3-compatible Clie									
000000FA	74	48	65	6C	6C	6F	20	68	61	6E	64	73	68	61	6B	65	20	77	61	73	20	66	6F	75	6E	tHello handshake was foun									
00000113	64	2E	20	46	69	64	64	6C	65	72	20	65	78	74	72	61	63	74	65	64	20	74	68	65	20	d. Fiddler extracted the									
0000012C	70	61	72	61	6D	65	74	65	72	73	20	62	65	6C	6F	77	2E	0A	0A	53	65	63	75	72	65	parameters below...Secure									
00000145	20	50	72	6F	74	6F	63	6F	6C	3A	20	54	4C	53	20	31	2E	33	0A	43	69	70	68	65	72	Protocol: TLS 1.3.Cipher									
0000015E	20	53	75	69	74	65	3A	20	54	4C	53	5F	41	45	53	5F	32	35	36	5F	47	43	4D	5F	53	Suite: TLS AES_256_GCM_S									
00000177	48	41	33	38	34	0A	0A	52	65	63	6F	72	64	20	4C	61	79	65	72	20	56	65	72	73	69	HA384..Record Layer Versi									
00000190	6F	6E	3A	20	33	2E	33	20	28	54	4C	53	2F	31	2E	32	29	0A	52	61	6E	64	6F	6D	3A	on: 3.3 (TLS/1.2).Random:									
000001A9	20	46	36	20	39	44	20	31	45	20	30	35	20	33	44	20	35	38	20	35	33	20	35	30	20	F6 9D 1E 05 3D 58 53 50									
000001C2	43	35	20	33	38	20	43	42	20	36	38	20	45	39	20	42	31	20	37	31	20	42	45	20	30	C5 38 A7 68 E9 B1 71 BE 0									
000001DB	32	20	37	37	20	41	37	20	46	41	20	41	42	20	33	46	20	43	43																

Second encryption algorithm

```
com:443 HTTP/1.1..Host: w
ww.higginbotham.com:443..
Connection: keep-alive..U
ser-Agent: Mozilla/5.0 (W
indows NT 10.0; Win64; x6
4) AppleWebKit/537.36 (KH
TML, like Gecko) Chrome/1
26.0.0.0 Safari/537.36...
..A SSLv3-compatible Clien
tHello handshake was foun
d. Fiddler extracted the
parameters below...Secure
Protocol: TLS 1.3.Cipher
Suite: TLS_AES_256_GCM_S
HA384..Record Layer Versi
on: 3.3 (TLS/1.2).Random:
f6 9D 1E 05 3D 58 53 50
C5 38 FB 68 E9 B1 71 BE 0
2 77 A7 FA AB 3F CC 1D 97
1B 4C AF AB 7A CD 69."Ti
me": 21:09-1972 08:33:18.
SessionID: 5E B3 4B 70 12
4D 2C CB 6A 5B 9A 63 89
```

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream
000004C9	36 37 20 39 38 20 39 46 20 38 42 20 38 38 20 44 39 20 41 30 20 43 36 20 31									67 98 9F 8B 88 D9 A0 C6 1
000004E2	35 20 43 43 20 33 37 20 42 42 20 35 31 20 39 31 20 31 44 20 33 30 20 43 45									5 CC 37 BB 51 91 1D 30 CE
000004FB	20 38 44 20 46 36 20 38 45 20 35 46 20 35 34 20 43 32 20 35 41 20 45 33 20									8D F6 8E 5F 54 C2 5A E3
00000514	38 39 20 32 43 20 37 42 20 38 39 20 41 41 20 42 42 20 43 35 20 41 46 20 45									39 2C 7B 89 AA BB C5 AF E
0000052D	39 20 35 36 20 31 30 20 42 38 20 31 35 20 35 37 20 39 34 20 41 31 20 41 31									9 56 10 B8 15 57 94 A1 A1
00000546	20 30 43 20 32 41 20 37 33 20 33 45 20 42 34 20 38 35 20 37 41 20 39 35 20									0C 2A 73 3E B4 85 7A 95
0000055F	34 34 20 33 34 20 30 37 20 31 39 20 43 38 20 33 33 20 33 46 20 45 43 20 39									44 34 07 19 C8 33 3F EC 9
00000578	43 20 34 35 20 34 36 20 30 42 20 38 34 20 43 39 20 31 43 20 32 32 20 45 31									C 45 46 0B 84 C9 1C 22 E1
00000591	20 33 43 20 39 46 20 38 37 20 37 31 20 41 33 20 35 42 20 43 33 20 37 32 20									3C 9F 87 71 A3 5B C3 72
000005AA	42 30 20 45 31 20 43 39 20 44 35 20 43 41 20 39 44 20 43 34 20 36 39 20 42									B0 E1 C9 D5 CA 9D C4 69 B
000005C3	30 20 30 35 20 35 42 20 39 44 20 45 32 20 41 34 20 32 42 20 35 36 20 46 35									0 05 5B 9D E2 A4 2B 56 F5
000005DC	20 42 46 20 38 32 20 38 34 20 36 37 20 39 43 20 43 36 20 36 35 20 35 46 20									BF 82 84 67 9C C6 65 5F
000005F5	38 32 20 33 45 20 42 32 20 46 30 20 42 42 20 46 35 20 33 42 20 41 46 20 30									82 3E B2 F0 BB F5 3B AF 0
0000060E	46 20 38 36 20 31 31 20 41 34 20 31 43 20 30 30 20 35 41 20 34 36 20 36 39									F 86 11 A4 1C 00 5A 46 69
00000627	20 44 39 20 35 37 20 41 46 20 33 31 20 43 36 20 43 32 20 46 34 20 46 30 20									D9 57 AF 31 C6 C2 F4 F0
00000640	33 38 20 46 37 20 45 43 20 39 37 20 36 41 20 32 31 20 37 38 20 43 34 20 41									38 F7 EC 97 6A 21 78 C4 A
00000659	38 20 41 41 20 42 34 20 38 35 20 43 45 20 43 39 20 38 35 20 36 37 20 44 37									8 AA B4 85 CE C9 85 67 D7
00000672	20 37 41 20 30 43 20 45 42 20 41 42 20 37 39 20 31 44 20 38 33 20 33 41 20									7A 0C EB AB 79 1D 83 3A
0000068B	42 32 20 33 42 20 36 30 20 44 36 20 41 33 20 43 32 20 30 31 20 38 37 20 41									B2 3B 60 D6 A3 C2 01 87 A
000006A4	37 20 46 37 20 43 43 20 42 38 20 41 30 20 45 35 20 31 45 20 31 46 20 35 35									7 F7 CC B8 A0 E5 1E 1F 55
000006BD	20 38 38 20 33 34 20 36 35 20 32 46 20 42 41 20 43 41 20 30 36 20 39 43 20									88 34 65 2F BA CA 06 9C
000006D6	33 42 20 31 37 20 39 44 20 41 36 20 31 31 20 42 32 20 33 39 20 39 46 20 33									3B 17 9D A6 11 B2 39 9F 3
000006EF	34 20 30 30 20 36 43 20 43 42 20 44 30 20 35 33 20 33 45 20 35 37 20 31 36									4 00 6C CB D0 53 3E 57 16

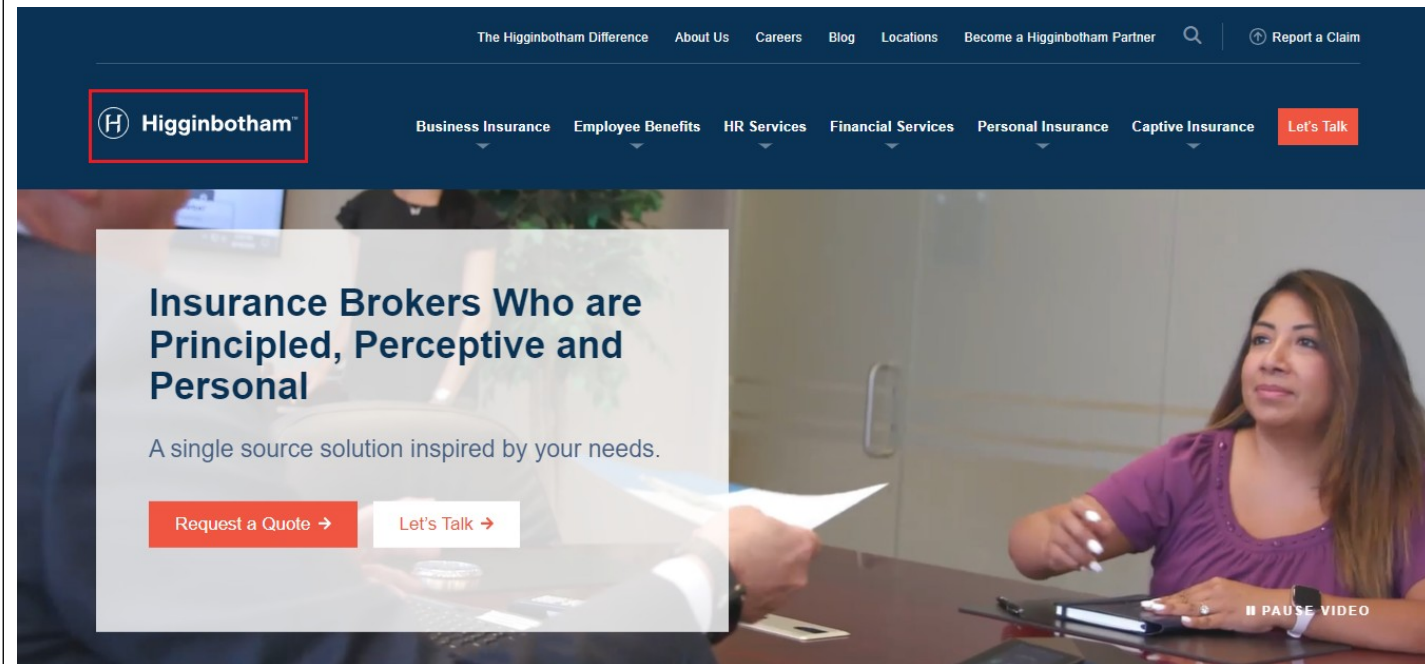
Source: Fiddler Capture

	<p>The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A <i>cryptographic hash function</i> is a function that computes a <i>message authentication code</i> from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.</p> <p>https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf</p> <p>The list of supported symmetric encryption algorithms has been pruned of all algorithms that are considered legacy. Those that remain are all Authenticated Encryption with Associated Data (AEAD) algorithms. The cipher suite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with both the key derivation function and <u>handshake message authentication code (MAC)</u>.</p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-4</p>
19. A system for a recursive security	The accused instrumentality utilizes a system for a recursive security protocol (e.g., TLS 1.3 security protocol) for protecting digital content (e.g., digital certificate related

protocol for protecting digital content, comprising a processor to execute instructions and a memory operable to store instructions for performing the steps of:

to the accused instrumentality), comprising a processor (e.g., a processor of the server of the accused instrumentality) to execute instructions and a memory (e.g., a memory of the server of the accused instrumentality) operable to store instructions.

The accused instrumentality utilizes TLS 1.3 security protocol (hereinafter “the standard”) for communicating content such as digital certificate, application data, etc., with a client. The standard provides a two-level encryption security. It encrypts a plaintext with a first encryption technique and generates a ciphertext. Further, it encrypts the ciphertext with a second encryption technique i.e., recursive encryption security.



<https://www.higginbotham.com/>

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

As shown below, the accused instrumentality utilizes a two-level algorithm security. It utilizes the SHA256RSA encryption algorithm as a first encryption algorithm i.e., signature encryption algorithm and the TLS_AES_256_GCM_SHA384 encryption algorithm as a second encryption algorithm i.e., AEAD encryption algorithm.

8	200	HTTP	Tunnel to	www.higginbotham.com:443	3,612
9	200	HTTPS	www.higginbotham.com	/	17,353
10	200	HTTPS	www.googleapis.com	/oauth2/v4/token	399
11	200	HTTPS	optimizationguide-p...	/v1:GetHints?	428
12	200	HTTP	Tunnel to	safebrowsing.google.com...	10,052
13	200	HTTPS	safebrowsing.google...	/safebrowsing/clientrepor...	32
14	200	HTTPS	www.higginbotham.com	/wp-includes/css/dist/bloc...	14,716
15	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	35,238
16	200	HTTP	Tunnel to	cdnjs.cloudflare.com:443	2,748
17	200	HTTP	Tunnel to	cdn.callrail.com:443	4,277
18	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	6,980
19	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	23,260
20	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	23,266
21	200	HTTPS	www.higginbotham.com	/wp-content/themes/orbit...	26,907
22	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	1,436
23	200	HTTPS	www.higginbotham.com	/wp-includes/js/jquery/qu...	31,396

TransformerHeadersTextViewSyntaxViewImageViewHexViewWebViewAuthCachingCookiesRawJSONXML

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3
Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate ==
[Version]
V3

[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com

[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US
Simple Name: GTS CA 1P5
DNS Name: GTS CA 1P5

Source: Fiddler Capture


```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B 8C 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 05 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 8C 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 8C AD B8 60 BF F2 BC 1E 43 34 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B 8C 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 05 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 8C 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 8C AD B8 60 BF F2 BC 1E 43 34 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

[Thumbprint]
129157F7A5852AD23DF17FE2934E6396DB6B0251

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption
algorithm

[Public Key]

Algorithm: RSA
Length: 2048

Key Blob: 30 82 01 0a 02 82 01 01 00 ca b8 66 5f 8a 37 94 14 e1 2f e3 49 cf d7 ad cb 20 8c 54 93 6b af 66 56 02 11 26 db 20 13 00 a0 ce fe ae e9 ac 8d 68 8f a0 56 fd ff b4 b1 4b 9c 54 ff 55 4a 95 76 b9 db 3c 77 5e 9a b1 dd 30 74 36 77 cb 92 12 df 17 62 6d 2a aa 74 3c f6 68 c8 34 8d eb 69 97 9e f2 43 02 fa d7 76 99 c0 bc b3 56 2d 17 ca 39 c8 00 5d 91 c8 1d 9b c9 70 c8 c1 b0 da 40 23 3e 9a fe a9 ed e4 60 ce 15 b0 af 43 a3 07 a5 81 25 dd c3 92 e7 72 9e 27 df bd 01 05 2a b4 60 b8 83 6b fa bc db 97 51 47 a8 2e d0 52 7f c2 0a 1c 02 96 d2 c7 fd 04 a5 7e 78 04 ee 59 ed 33 9a e4 cc 42 bb 87 36 95 ac 4b 9b 6b 4f 96 4c ed 20 f7 bd 71 4b bc d8 91 81 3c be 56 c8 05 a6 e1 7f 66 85 88 46 91 96 a3 e4 e8 bb 45 f5 b8 78 25 4b 02 cb 23 e6 54 94 ed 94 f6 87 3d 29 55 31 eb 59 e9 a1 54 4c 50 b0 74 2c a7 b6 1e 7f 23 02 03 01 00 01

Parameters: 05 00

[Extensions]

* Key Usage(2.5.29.15):
Digital Signature, Key Encipherment (a0)

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm
00000019	63 6F 6D 3A 34 34 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 77									com:443 HTTP/1.1..Host: w
00000032	77 77 2E 68 69 67 67 69 6E 62 6F 74 68 61 6D 2E 63 6F 6D 3A 34 34 33 0D 0A									ww.migginbotham.com:443..
0000004B	43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 6B 65 65 70 2D 61 6C 69 76 65 0D 0A 55									Connection: keep-alive..U
00000064	73 65 72 2D 41 67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 57									ser-Agent: Mozilla/5.0 (W
0000007D	69 6E 64 6F 77 73 20 4E 54 20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36									indows NT10.0; Win64; x6
00000096	34 29 20 41 70 70 6C 65 57 65 62 4B 69 74 2F 35 33 37 2E 33 36 20 28 4B 48									4) AppleWebKit/537.36 (KH
000000AF	54 4D 4C 2C 20 6C 69 6B 65 20 47 65 63 6B 6F 29 20 43 68 72 6F 6D 65 2F 31									TML, like Gecko) Chrome/1
000000C8	32 36 2E 30 2E 30 2E 30 20 53 61 66 61 72 69 2F 35 33 37 2E 33 36 0D 0A 0D									26.0.0.0 Safari/537.36...
000000E1	0A 41 20 53 53 4C 76 33 2D 63 6F 6D 70 61 74 69 62 6C 65 20 43 6C 69 65 6E									.A SSLv3-compatible Clie
000000FA	74 48 65 6C 6C 6F 20 68 61 6E 64 73 68 61 6B 65 20 77 61 73 20 66 6F 75 6E									tHello handshake was foun
00000113	64 2E 20 46 69 64 64 6C 65 72 20 65 78 74 72 61 63 74 65 64 20 74 68 65 20									d. Fiddler extracted the
0000012C	70 61 72 61 6D 65 74 65 72 73 20 62 65 6C 6F 77 2E 0A 0A 53 65 63 75 72 65									parameters below...Secure
00000145	20 50 72 6F 74 6F 63 6F 6C 3A 20 54 4C 53 20 31 2E 33 0A 43 69 70 68 65 72									Protocol: TLS 1.3.Cipher
0000015E	20 53 75 69 74 65 3A 20 54 4C 53 5F 41 45 53 5F 32 35 36 5F 47 43 4D 5F 53									Suite: TLS_AES_256_GCM_S
00000177	48 41 33 38 34 0A 0A 52 65 63 6F 72 64 20 4C 61 79 65 72 20 56 65 72 73 69									HA384..Record Layer Versi
00000190	6F 6E 3A 20 33 2E 33 20 28 54 4C 53 2F 31 2E 32 29 0A 52 61 6E 64 6F 6D 3A									on: 3.3 (TLS/1.2).Random:
000001A9	20 46 36 20 39 44 20 31 45 20 30 35 20 33 44 20 35 38 20 35 33 20 35 30 20									F6 9D 1E 05 3D 58 53 50
000001C2	43 35 20 33 38 20 43 42 20 36 38 20 45 39 20 42 31 20 37 31 20 42 45 20 30									C5 38 CB 68 E9 B1 71 BE 0
000001DB	32 20 37 37 20 41 37 20 46 41 20 41 42 20 33 46 20 43 43 20 31 44 20 39 37									2 77 A7 FA AB 3F CC 1D 97
000001F4	20 31 42 20 34 43 20 41 46 20 41 42 20 37 41 20 43 44 20 36 39 0A 22 54 69									1B 4C AF AB 7A CD 69."Ti
0000020D	6D 65 22 3A 20 32 31 2D 30 39 2D 31 39 37 32 20 30 38 3A 33 33 3A 31 38 0A									me": 21-09-1972 08:33:18.
00000226	53 65 73 73 69 6F 6E 49 44 3A 20 35 45 20 42 33 20 34 42 20 37 30 20 31 32									SessionID: 5E B3 4B 70 12
0000023F	20 34 44 20 32 43 20 43 42 20 36 41 20 35 42 20 39 41 20 36 33 20 38 39 20									4D 2C CB 6A 5B 9A 63 89

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream																								
000004C9	36	37	20	39	38	20	39	46	20	38	42	20	38	38	20	44	39	20	41	30	20	43	36	20	31	67	98	9F	8B	88	D9	A0	C6	1
000004E2	35	20	43	43	20	33	37	20	42	42	20	35	31	20	39	31	20	31	44	20	33	30	20	43	45	5	CC	37	BB	51	91	1D	30	CE
000004FB	20	38	44	20	46	36	20	38	45	20	35	46	20	35	34	20	43	32	20	35	41	20	45	33	20	8D	F6	8E	5F	54	C2	5A	E3	
00000514	38	39	20	32	43	20	37	42	20	38	39	20	41	41	20	42	42	20	43	35	20	41	46	20	45	39	2C	7B	89	AA	BB	C5	AF	E
0000052D	39	20	35	36	20	31	30	20	42	38	20	31	35	20	35	37	20	39	34	20	41	31	20	41	31	9	56	10	B8	15	57	94	A1	A1
00000546	20	30	43	20	32	41	20	37	33	20	33	45	20	42	34	20	38	35	20	37	41	20	39	35	20	0C	2A	73	3E	B4	85	7A	95	
0000055F	34	34	20	33	34	20	30	37	20	31	39	20	43	38	20	33	33	20	33	46	20	45	43	20	39	44	34	07	19	C8	33	3F	EC	9
00000578	43	20	34	35	20	34	36	20	30	42	20	38	34	20	43	39	20	31	43	20	32	32	20	45	31	C	45	46	0B	84	C9	1C	22	E1
00000591	20	33	43	20	39	46	20	38	37	20	37	31	20	41	33	20	35	42	20	43	33	20	37	32	20	3C	9F	87	71	A3	5B	C3	72	
000005AA	42	30	20	45	31	20	43	39	20	44	35	20	43	41	20	39	44	20	43	34	20	36	39	20	42	B0	E1	C9	D5	CA	9D	C4	69	B
000005C3	30	20	30	35	20	35	42	20	39	44	20	45	32	20	41	34	20	32	42	20	35	36	20	46	35	0	05	5B	9D	E2	A4	2B	56	F5
000005DC	20	42	46	20	38	32	20	38	34	20	36	37	20	39	43	20	43	36	20	36	35	20	35	46	20	BF	82	84	67	9C	C6	65	5F	
000005F5	38	32	20	33	45	20	42	32	20	46	30	20	42	42	20	46	35	20	33	42	20	41	46	20	30	82	3E	B2	F0	BB	F5	3B	AF	0
0000060E	46	20	38	36	20	31	31	20	41	34	20	31	43	20	30	30	20	35	41	20	34	36	20	36	39	F	86	11	A4	1C	00	5A	46	69
00000627	20	44	39	20	35	37	20	41	46	20	33	31	20	43	36	20	43	32	20	46	34	20	46	30	20	D9	57	AF	31	C6	C2	F4	F0	
00000640	33	38	20	46	37	20	45	43	20	39	37	20	36	41	20	32	31	20	37	38	20	43	34	20	41	38	F7	EC	97	6A	21	78	C4	A
00000659	38	20	41	41	20	42	34	20	38	35	20	43	45	20	43	39	20	38	35	20	36	37	20	44	37	8	AA	B4	85	CE	C9	85	67	D7
00000672	20	37	41	20	30	43	20	45	42	20	41	42	20	37	39	20	31	44	20	38	33	20	33	41	20	7A	0C	EB	AB	79	1D	83	3A	
0000068B	42	32	20	33	42	20	36	30	20	44	36	20	41	33	20	43	32	20	30	31	20	38	37	20	41	B2	3B	60	D6	A3	C2	01	87	A
000006A4	37	20	46	37	20	43	43	20	42	38	20	41	30	20	45	35	20	31	45	20	31	46	20	35	35	7	F7	CC	B8	A0	E5	1E	1F	55
000006BD	20	38	38	20	33	34	20	36	35	20	32	46	20	42	41	20	43	41	20	30	36	20	39	43	20	88	34	65	2F	BA	CA	06	9C	
000006D6	33	42	20	31	37	20	39	44	20	41	36	20	31	31	20	42	32	20	33	39	20	39	46	20	33	3B	17	9D	A6	11	B2	39	9F	3
000006EF	34	20	30	30	20	36	43	20	43	42	20	44	30	20	35	33	20	33	45	20	35	37	20	31	36	4	00	6C	CB	D0	53	3E	57	16

Source: Fiddler Capture

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3

Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate ==

[Version]
V3

[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com

[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US

Second decryption
algorithm

Source: Fiddler Capture

As shown below, the server of the accused instrumentality comprises a processor to

execute instructions and a memory storage to store instructions for performing the operations defined by the standard.

```
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret  empty
ec_point_formats  uncompressed [0x0]
status_request  OCSP - Implicit Responder
psk_key_exchange_modes  01 01
renegotiation_info  00
0xfe0d  00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 DD 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 88 BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 85 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name  www.higginbotham.com
crease (0x3a3a)  00
```

Source: Fiddler Capture



Tech Accelerator

Server hardware guide: Architecture, products and management



2. Processor

The CPU -- or simply processor -- is a complex micro-circuitry device that serves as the foundation of all computer operations. It supports hundreds of possible commands hardwired into hundreds of millions of transistors to process low-level software instructions -- microcode -- and data and derive a desired logical or mathematical result. The processor works closely with memory, which both holds the software instructions and data to be processed as well as the results or output of those processor operations.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>



Tech Accelerator

Server hardware guide: Architecture, products and management

3. Random access memory



RAM is the main type of memory in a computing system.

RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as

data to be moved to a storage device. Thus, RAM works very

closely with the processor and must match the processor's

incredible speed and performance. This kind of fast memory

is usually termed dynamic RAM, and several DRAM

variations are available for servers.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>



Tech Accelerator

Server hardware guide: Architecture, products and management



4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

<https://www.techtargget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

<https://datatracker.ietf.org/doc/html/rfc8446#>

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<----- [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

Second
encryption

- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.

First
encryption

- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.

- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Introduction

The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties:

- Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated.

Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK).

- Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques.
- Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection.

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^{14} bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [\[RFC5116\], Section 4](#)). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>This specification defines the following cipher suites for use with TLS 1.3.</p> <table border="1"> <thead> <tr> <th>Description</th><th>Value</th></tr> </thead> <tbody> <tr> <td>TLS_AES_128_GCM_SHA256</td><td>{0x13,0x01}</td></tr> <tr> <td>TLS_AES_256_GCM_SHA384</td><td>{0x13,0x02}</td></tr> <tr> <td>TLS_CHACHA20_POLY1305_SHA256</td><td>{0x13,0x03}</td></tr> <tr> <td>TLS_AES_128_CCM_SHA256</td><td>{0x13,0x04}</td></tr> <tr> <td>TLS_AES_128_CCM_8_SHA256</td><td>{0x13,0x05}</td></tr> </tbody> </table> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>	Description	Value	TLS_AES_128_GCM_SHA256	{0x13,0x01}	TLS_AES_256_GCM_SHA384	{0x13,0x02}	TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}	TLS_AES_128_CCM_SHA256	{0x13,0x04}	TLS_AES_128_CCM_8_SHA256	{0x13,0x05}
Description	Value												
TLS_AES_128_GCM_SHA256	{0x13,0x01}												
TLS_AES_256_GCM_SHA384	{0x13,0x02}												
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}												
TLS_AES_128_CCM_SHA256	{0x13,0x04}												
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}												
encrypting a bit stream with a first encryption algorithm;	<p>The standard practices encrypting a bitstream (e.g., bitstream of digital certificate) with a first encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA encryption algorithm).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext.</p>												

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

As shown below, the accused instrumentality discloses the signature encryption algorithm.

8200HTTP Tunnel to www.higginbotham.com:4433,612

9200HTTPS www.higginbotham.com / 17,353

10200HTTPS www.googleapis.com /oauth2/v4/token 399

11200HTTPS optimizationguide-p... /v1:GetHints? 428

12200HTTP Tunnel to safebrowsing.google.com... 10,052

13200HTTPS safebrowsing.google.com /safebrowsing/clientrepor... 32

14200HTTPS www.higginbotham.com /wp-includes/css/dist/bloc... 14,716

15200HTTPS www.higginbotham.com /wp-content/cache/min/1/... 35,238

16200HTTP Tunnel to cdnjs.cloudflare.com:443 2,748

17200HTTP Tunnel to cdn.cloudflare.com:443 4,277

18200HTTPS www.higginbotham.com /wp-content/cache/min/1/... 6,980

19200HTTPS www.higginbotham.com /wp-content/cache/min/1/... 23,260

20200HTTPS www.higginbotham.com /wp-content/cache/min/1/... 23,266

21200HTTPS www.higginbotham.com /wp-content/themes/orbit... 26,907

22200HTTPS www.higginbotham.com /wp-content/cache/min/1/... 1,436

23200HTTPS www.higginbotham.com /wp-includes/js/jquery/jqu... 31,396

Transformer Headers TextView SyntaxView ImageView HexView WebView Auth Caching Cookies Raw JSON XML

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3
Cipher Suite: TLS_AES_256_GCM_SHA384

-- Server Certificate -----
[Version]
V3
[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com
[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US
Simple Name: GTS CA 1P5
DNS Name: GTS CA 1P5

Source: Fiddler Capture


```

SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 C8 B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B 8C 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 FB 9F E9 09 14 5A C3 B6 5A 5B 98 D2 05 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD D0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 4B 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 8C 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 8C AD B8 60 BF F2 BC 1E 43 24 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00

```

Source: Fiddler Capture

```

SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 C8 B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B 8C 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 FB 9F E9 09 14 5A C3 B6 5A 5B 98 D2 05 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD D0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 4B 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 8C 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 8C AD B8 60 BF F2 BC 1E 43 24 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00

```

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
     | {CertificateVerify*}
     v {Finished}
       [Application Data] ----->
<----- [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.

- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A First encryption "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.

- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```




<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>Introduction</p> <p>The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties:</p> <ul style="list-style-type: none"> - Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK). - Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques. - Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection. <p>https://datatracker.ietf.org/doc/html/rfc8446</p>
<p>associating a first decryption algorithm with the encrypted bit stream;</p>	<p>The standard practices associating a first decryption algorithm (e.g., signature decryption algorithm i.e., SHA256RSA decryption algorithm) with the encrypted bit stream (e.g., encrypted certificate with signature encryption algorithm).</p> <p>The standard practices providing a two-level encryption security for data</p>

communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext.

The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate.

Security overview



This page is secure (valid HTTPS).

- Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

View certificate
- Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.
- Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

As shown below, the accused instrumentality discloses the signature decryption algorithm.

[Thumbprint]
129157F7A5852AD23DF17FE2934E6396DB6B0251

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption
algorithm

[Public Key]

Algorithm: RSA
Length: 2048

Key Blob: 30 82 01 0a 02 82 01 01 00 ca b8 66 5f 8a 37 94 14 e1 2f e3 49 cf d7 ad cb 20 8c 54 93 6b af 66 56 02 11 26 db 20 13 00 a0 ce fe ae e9 ac 8d 68 8f a0 56 fd ff b4 b1 4b 9c 54 ff 55 4a 95 76 b9 db 3c 77 5e 9a b1 dd 30 74 36 77 cb 92 12 df 17 62 6d 2a aa 74 3c f6 68 c8 34 8d eb 69 97 9e f2 43 02 fa d7 76 99 c0 bc b3 56 2d 17 ca 39 c8 00 5d 91 c8 1d 9b c9 70 c8 c1 b0 da 40 23 3e 9a fe a9 ed e4 60 ce 15 b0 af 43 a3 07 a5 81 25 dd c3 92 e7 72 9e 27 df bd 01 05 2a b4 60 b8 83 6b fa bc db 97 51 47 a8 2e d0 52 7f c2 0a 1c 02 96 d2 c7 fd 04 a5 7e 78 04 ee 59 ed 33 9a e4 cc 42 bb 87 36 95 ac 4b 9b 6b 4f 96 4c ed 20 f7 bd 71 4b bc d8 91 81 3c be 56 c8 05 a6 e1 7f 66 85 88 46 91 96 a3 e4 e8 bb 45 f5 b8 78 25 4b 02 cb 23 e6 54 94 ed 94 f6 87 3d 29 55 31 eb 59 e9 a1 54 4c 50 b0 74 2c a7 b6 1e 7f 23 02 03 01 00 01

Parameters: 05 00

[Extensions]

* Key Usage(2.5.29.15):

Digital Signature, Key Encipherment (a0)

Source: Fiddler Capture

OID description

First decryption algorithm identifier

OID:	{iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-1(1) sha256WithRSAEncryption(11)}	(ASN.1 notation)
	1.2.840.113549.1.1.11	(dot notation)
	/ISO/Member-Body/US/113549/1/1/11	(OID-IRI notation)

Description:

Public-Key Cryptography Standards (PKCS) #1 version 1.5 signature algorithm with Secure Hash Algorithm 256 (SHA256) and Rivest, Shamir and Adleman (RSA) encryption

<http://oid-info.com/get/1.2.840.113549.1.1.11>

-- When the following OIDs are used in an AlgorithmIdentifier, the
-- parameters MUST be present and MUST be NULL.

sha224WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 14 }

sha256WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 11 }

sha384WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 12 }

sha512WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 13 }

<https://www.ietf.org/rfc/rfc4055.txt>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<-----> [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<-----> [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.
- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.
- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A First encryption "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.
- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

First
decryption
algorithm
information

The algorithm field specifies the signature algorithm used (see [Section 4.2.3](#) for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in [Section 4.4.1](#), namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$




First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

	<p>The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A <i>cryptographic hash function</i> is a function that computes a <i>message authentication code</i> from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.</p> <p>https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf</p>
<p>encrypting both the encrypted bit stream and the first decryption algorithm with a second encryption algorithm to yield a second bit stream;</p>	<p>The standard practices encrypting both the encrypted bit stream (e.g., encrypted digital certificate) and the first decryption algorithm (e.g., signature decryption algorithm) with a second encryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) to yield a second bit stream (e.g., TLS ciphertext bitstream).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</p> <p>The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it.</p>

The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.

Security overview

This page is secure (valid HTTPS).

- Certificate - valid and trusted
The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.
[View certificate](#)
- Connection - secure connection settings
The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.
- Resources - all served securely
All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm
00000019	63 6F 6D 3A 34 34 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 77									con:443 HTTP/1.1..Host: w
00000032	77 77 2E 68 69 67 67 69 6E 62 6F 74 68 61 6D 2E 63 6F 6D 3A 34 34 33 0D 0A									ww.higginbotham.com:443..
0000004B	43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 6B 65 65 70 2D 61 6C 69 76 65 0D 0A 55									Connection: keep-alive..U
00000064	73 65 72 2D 41 67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 57									ser-Agent: Mozilla/5.0 (W
0000007D	69 6E 64 6F 77 73 20 4E 54 20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36									indows NT 10.0; Win64; x6
00000096	34 29 20 41 70 70 6C 65 57 65 62 4B 69 74 2F 35 33 37 2E 33 36 20 28 4B 48									4) AppleWebKit/537.36 (KH
000000AF	54 4D 4C 2C 20 6C 69 6B 65 20 47 65 63 6B 6F 29 20 43 68 72 6F 6D 65 2F 31									TML, like Gecko) Chrome/1
000000C8	32 36 2E 30 2E 30 2E 30 20 53 61 66 61 72 69 2F 35 33 37 2E 33 36 0D 0A 0D									26.0.0.0 Safari/537.36...
000000E1	0A 41 20 53 53 4C 76 33 2D 63 6F 6D 70 61 74 69 62 6C 65 20 43 6C 69 65 6E									.A SSLv3-compatible Clie
000000FA	74 48 65 6C 6C 6F 20 68 61 6E 64 73 68 61 6B 65 20 77 61 73 20 66 6F 75 6E									tHello handshake was foun
00000113	64 2E 20 46 69 64 64 6C 65 72 20 65 78 74 72 61 63 74 65 64 20 74 68 65 20									d. Fiddler extracted the
0000012C	70 61 72 61 6D 65 74 65 72 73 20 62 65 6C 6F 77 2E 0A 0A 53 65 63 75 72 65									parameters below...Secure
00000145	20 50 72 6F 74 6F 63 6F 6C 3A 20 54 4C 53 20 31 2E 33 0A 43 69 70 68 65 72									Protocol: TLS 1.3.Cipher
0000015E	20 53 75 69 74 65 3A 20 54 4C 53 5F 41 45 53 5F 32 35 36 5F 47 43 4D 5F 53									Suite: TLS_AES_256_GCM_S
00000177	48 41 33 38 34 0A 0A 52 65 63 6F 72 64 20 4C 61 79 65 72 20 56 65 72 73 69									HA384..Record Layer Versi
00000190	6F 6E 3A 20 33 2E 33 20 28 54 4C 53 2F 31 2E 32 29 0A 52 61 6E 64 6F 6D 3A									on: 3.3 (TLS/1.2).Random:
000001A9	20 46 36 20 39 44 20 31 45 20 30 35 20 33 44 20 35 38 20 35 33 20 35 30 20									F6 9D 1E 05 3D 58 53 50
000001C2	43 35 20 33 38 20 43 42 20 36 38 20 45 39 20 42 31 20 37 31 20 42 45 20 30									C5 38 CB 68 E9 B1 71 BE 0
000001DB	32 20 37 37 20 41 37 20 46 41 20 41 42 20 33 46 20 43 43 20 31 44 20 39 37									2 77 A7 FA AB 3F CC 1D 97
000001F4	20 31 42 20 34 43 20 41 46 20 41 42 20 37 41 20 43 44 20 36 39 0A 22 54 69									1B 4C AF AB 7A CD 69."Ti
0000020D	6D 65 22 3A 20 32 31 2D 30 39 2D 31 39 37 32 20 30 38 3A 33 33 3A 31 38 0A									me": 21-09-1972 08:33:18.
00000226	53 65 73 73 69 6F 6E 49 44 3A 20 35 45 20 42 33 20 34 42 20 37 30 20 31 32									SessionID: 5E B3 4B 70 12
0000023F	20 34 44 20 32 43 20 43 42 20 36 41 20 35 42 20 39 41 20 36 33 20 38 39 20									4D 2C CB 6A 5B 9A 63 89

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream
000004C9	36 37 20 39 38 20 39 46 20 38 42 20 38 38 20 44 39 20 41 30 20 43 36 20 31									67 98 9F 8B 88 D9 A0 C6 1
000004E2	35 20 43 43 20 33 37 20 42 42 20 35 31 20 39 31 20 31 44 20 33 30 20 43 45									5 CC 37 BB 51 91 1D 30 CE
000004FB	20 38 44 20 46 36 20 38 45 20 35 46 20 35 34 20 43 32 20 35 41 20 45 33 20									8D F6 8E 5F 54 C2 5A E3
00000514	38 39 20 32 43 20 37 42 20 38 39 20 41 41 20 42 42 20 43 35 20 41 46 20 45									39 2C 7B 89 AA BB C5 AF E
0000052D	39 20 35 36 20 31 30 20 42 38 20 31 35 20 35 37 20 39 34 20 41 31 20 41 31									9 56 10 B8 15 57 94 A1 A1
00000546	20 30 43 20 32 41 20 37 33 20 33 45 20 42 34 20 38 35 20 37 41 20 39 35 20									0C 2A 73 3E B4 85 7A 95
0000055F	34 34 20 33 34 20 30 37 20 31 39 20 43 38 20 33 33 20 33 46 20 45 43 20 39									44 34 07 19 C8 33 3F EC 9
00000578	43 20 34 35 20 34 36 20 30 42 20 38 34 20 43 39 20 31 43 20 32 32 20 45 31									C 45 46 0B 84 C9 1C 22 E1
00000591	20 33 43 20 39 46 20 38 37 20 37 31 20 41 33 20 35 42 20 43 33 20 37 32 20									3C 9F 87 71 A3 5B C3 72
000005AA	42 30 20 45 31 20 43 39 20 44 35 20 43 41 20 39 44 20 43 34 20 36 39 20 42									B0 E1 C9 D5 CA 9D C4 69 B
000005C3	30 20 30 35 20 35 42 20 39 44 20 45 32 20 41 34 20 32 42 20 35 36 20 46 35									0 05 5B 9D E2 A4 2B 56 F5
000005DC	20 42 46 20 38 32 20 38 34 20 36 37 20 39 43 20 43 36 20 36 35 20 35 46 20									BF 82 84 67 9C C6 65 5F
000005F5	38 32 20 33 45 20 42 32 20 46 30 20 42 42 20 46 35 20 33 42 20 41 46 20 30									82 3E B2 F0 BB F5 3B AF 0
0000060E	46 20 38 36 20 31 31 20 41 34 20 30 31 43 20 30 30 20 35 41 20 34 36 20 36 39									F 86 11 A4 1C 00 5A 46 69
00000627	20 44 39 20 35 37 20 41 46 20 33 31 20 43 36 20 43 32 20 46 34 20 46 30 20									D9 57 AF 31 C6 C2 F4 F0
00000640	33 38 20 46 37 20 45 43 20 39 37 20 36 41 20 32 31 20 37 38 20 43 34 20 41									38 F7 EC 97 6A 21 78 C4 A
00000659	38 20 41 41 20 42 34 20 38 35 20 43 45 20 43 39 20 38 35 20 36 37 20 44 37									8 AA B4 85 CE C9 85 67 D7
00000672	20 37 41 20 30 43 20 45 42 20 41 42 20 37 39 20 31 44 20 38 33 20 33 41 20									7A 0C EB AB 79 1D 83 3A
0000068B	42 32 20 33 42 20 36 30 20 44 36 20 41 33 20 43 32 20 30 31 20 38 37 20 41									B2 3B 60 D6 A3 C2 01 87 A
000006A4	37 20 46 37 20 43 43 20 42 38 20 41 30 20 45 35 20 31 45 20 31 46 20 35 35									7 F7 CC B8 A0 E5 1E 1F 55
000006BD	20 38 38 20 33 34 20 36 35 20 32 46 20 42 41 20 43 41 20 30 36 20 39 43 20									88 34 65 2F BA CA 06 9C
000006D6	33 42 20 31 37 20 39 44 20 41 36 20 31 31 20 42 32 20 33 39 20 39 46 20 33									3B 17 9D A6 11 B2 39 9F 3
000006EF	34 20 30 30 20 36 43 20 43 42 20 44 30 20 35 33 20 33 45 20 35 37 20 31 36									4 00 6C CB D0 53 3E 57 16

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext

handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^{14} bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding `TLSPlaintext.length` due to the inclusion of `TLSInnerPlaintext.type` and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (`iv_length`) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [\[RFC5116\], Section 4](#)). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

All handshake messages after the ServerHello are now encrypted.
The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

Client

Server

Key ^ ClientHello

Exch | + key_share*

```
| + signature_algorithms*
```

```
| + psk_key_exchange_modes*
```

$$v + \text{pre_shared_key}^* \rightarrow$$

ServerHello ^ Key

+ key_share* | Exch

$$+ \text{pre_shared_key}^* \quad v$$

```
{EncryptedExtensions}
```

```
{CertificateRequest*}
```

```
{Certificate*}
```

```
{CertificateVerify*} | Auth
```

```
{Finished}
```

[Application Data*]

Digital Content

$$\wedge \{ \text{Certificate}^* \}$$

Auth | {CertificateVerify*}

v {Finished}

[Application Data]

----->

```
<-----> [Application Data]
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

Second
encryption

- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.

First
encryption

- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.

- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

First
decryption
algorithm
information

The algorithm field specifies the signature algorithm used (see [Section 4.2.3](#) for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in [Section 4.4.1](#), namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>The <u>"extension_data" field of these extensions contains a SignatureSchemeList value:</u></p> <pre>enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), }</pre> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>associating a second decryption algorithm with the second bit stream.</p>	<p>The standard practices associating a second decryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) with the second bit stream (e.g., TLS ciphertext bitstream).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</p>

The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

Fiddler - Web Sessions						Transformer Headers TextView SyntaxView ImageView HexView WebView Auth Caching Cookies Raw JSON XML									
8	200	HTTP	Tunnel to	www.higginbotham.com:443	3,612	Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.									
9	200	HTTPS	www.higginbotham.com...	/	17,353	Secure Protocol: TLS 1.3 Cipher Suite: TLS_AES_256_GCM_SHA384									
10	200	HTTPS	www.googleapis.com	/oauth2/v4/token	399	== Server Certificate == [Version] V3									
11	200	HTTPS	optimizationguide-p...	/v1:GetHints?	428	[Subject] CN=higginbotham.com Simple Name: higginbotham.com DNS Name: higginbotham.com									
12	200	HTTP	Tunnel to	safebrowsing.google.com...	10,052	[Issuer] CN=GTS CA 1P5, O=Google Trust Services LLC, C=US Simple Name: GTS CA 1P5 DNS Name: GTS CA 1P5									
13	200	HTTPS	safebrowsing.googl...	/safebrowsing/clientrepor...	32										
14	200	HTTPS	www.higginbotham.com...	/wp-includes/css/dist/bloc...	14,716										
15	200	HTTPS	www.higginbotham.com...	/wp-content/cache/min/1/...	35,238										
16	200	HTTP	Tunnel to	cdnjs.cloudflare.com:443	2,748										
17	200	HTTP	Tunnel to	cdn.callrail.com:443	4,277										
18	200	HTTPS	www.higginbotham.com...	/wp-content/cache/min/1/...	6,980										
19	200	HTTPS	www.higginbotham.com...	/wp-content/cache/min/1/...	23,260										
20	200	HTTPS	www.higginbotham.com...	/wp-content/cache/min/1/...	23,266										
21	200	HTTPS	www.higginbotham.com...	/wp-content/themes/orbit...	26,907										
22	200	HTTPS	www.higginbotham.com...	/wp-content/cache/min/1/...	1,436										
23	200	HTTPS	www.higginbotham.com...	/wp-includes/js/jquery/qu...	31,396										

Source: Fiddler Capture

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3

Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate =====
[Version]
V3

Second decryption
algorithm

[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com

[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream
000004C9	36 37 20 39 38 20 39 46 20 38 42 20 38 38 20 44 39 20 41 30 20 43 36 20 31									67 98 9F 8B 88 D9 A0 C6 1
000004E2	35 20 43 43 20 33 37 20 42 42 20 35 31 20 39 31 20 31 44 20 33 30 20 43 45									5 CC 37 BB 51 91 1D 30 CE
000004FB	20 38 44 20 46 36 20 38 45 20 35 46 20 35 34 20 43 32 20 35 41 20 45 33 20									8D F6 8E 5F 54 C2 5A E3
00000514	38 39 20 32 43 20 37 42 20 38 39 20 41 41 20 42 42 20 43 35 20 41 46 20 45									39 2C 7B 89 AA BB C5 AF E
0000052D	39 20 35 36 20 31 30 20 42 38 20 31 35 20 35 37 20 39 34 20 41 31 20 41 31									9 56 10 B8 15 57 94 A1 A1
00000546	20 30 43 20 32 41 20 37 33 20 33 45 20 42 34 20 38 35 20 37 41 20 39 35 20									0C 2A 73 3E B4 85 7A 95
0000055F	34 34 20 33 34 20 30 37 20 31 39 20 43 38 20 33 33 20 33 46 20 45 43 20 39									44 34 07 19 C8 33 3F EC 9
00000578	43 20 34 35 20 34 36 20 30 42 20 38 34 20 43 39 20 31 43 20 32 32 20 45 31									C 45 46 0B 84 C9 1C 22 E1
00000591	20 33 43 20 39 46 20 38 37 20 37 31 20 41 33 20 35 42 20 43 33 20 37 32 20									3C 9F 87 71 A3 5B C3 72
000005AA	42 30 20 45 31 20 43 39 20 44 35 20 43 41 20 39 44 20 43 34 20 36 39 20 42									B0 E1 C9 D5 CA 9D C4 69 B
000005C3	30 20 30 35 20 35 42 20 39 44 20 45 32 20 41 34 20 32 42 20 35 36 20 46 35									0 05 5B 9D E2 A4 2B 56 F5
000005DC	20 42 46 20 38 32 20 38 34 20 36 37 20 39 43 20 43 36 20 36 35 20 35 46 20									BF 82 84 67 9C C6 65 5F
000005F5	38 32 20 33 45 20 42 32 20 46 30 20 42 42 20 46 35 20 33 42 20 41 46 20 30									82 3E B2 F0 BB F5 3B AF 0
0000060E	46 20 38 36 20 31 31 20 41 34 20 31 43 20 30 30 20 35 41 20 34 36 20 36 39									F 86 11 A4 1C 00 5A 46 69
00000627	20 44 39 20 35 37 20 41 46 20 33 31 20 43 36 20 43 32 20 46 34 20 46 30 20									D9 57 AF 31 C6 C2 F4 F0
00000640	33 38 20 46 37 20 45 43 20 39 37 20 36 41 20 32 31 20 37 38 20 43 34 20 41									38 F7 EC 97 6A 21 78 C4 A
00000659	38 20 41 41 20 42 34 20 38 35 20 43 45 20 43 39 20 38 35 20 36 37 20 44 37									8 AA B4 85 CE C9 85 67 D7
00000672	20 37 41 20 30 43 20 45 42 20 41 42 20 37 39 20 31 44 20 38 33 20 33 41 20									7A 0C EB AB 79 1D 83 3A
0000068B	42 32 20 33 42 20 36 30 20 44 36 20 41 33 20 43 32 20 30 31 20 38 37 20 41									B2 3B 60 D6 A3 C2 01 87 A
000006A4	37 20 46 37 20 43 43 20 42 38 20 41 30 20 45 35 20 31 45 20 31 46 20 35 35									7 F7 CC B8 A0 E5 1E 1F 55
000006BD	20 38 38 20 33 34 20 36 35 20 32 46 20 42 41 20 43 41 20 30 36 20 39 43 20									88 34 65 2F BA CA 06 9C
000006D6	33 42 20 31 37 20 39 44 20 41 36 20 31 31 20 42 32 20 33 39 20 39 46 20 33									3B 17 9D A6 11 B2 39 9F 3
000006EF	34 20 30 30 20 36 43 20 43 42 20 44 30 20 35 33 20 33 45 20 35 37 20 31 36									4 00 6C CB D0 53 3E 57 16

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS

communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^{14} bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.2. Authenticated Decryption

Second decryption algorithm

The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [\[RFC5116\], Section 4](#)). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block.

These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<-----> [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>The <u>"extension_data" field of these extensions contains a SignatureSchemeList value:</u></p> <pre> enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), </pre> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>20. The system of claim 19, further operable for decrypting the first bit stream and the second</p>	<p>The standard further discloses decrypting the first bit stream (e.g., encrypted digital certificate with signature encryption algorithm i.e., SHA-256 RSA, etc.) and the second bit stream (e.g., a second-level encryption with AEAD encryption algorithm such as TLS AES 256 GCM SHA384, etc.) with the first associated decryption</p>

<p>bit stream with the first associated decryption algorithm and the second associated decryption algorithm wherein the decryption is accomplished by a target unit.</p>	<p>algorithm (e.g., signature decryption algorithm i.e., SHA-256 RSA, etc.) and the second associated decryption algorithm (e.g., cipher suit selected from one of the AEAD decryption algorithms such as TLS_AES_256_GCM_SHA384, etc.) wherein the decryption is accomplished by a target unit (e.g., a server of the accused instrumentality).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</p> <p>The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.</p>
--	---

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

						Transformer	Headers	TextView	SyntaxView	ImageView	HexView	WebView	Auth	Caching	Cookies	Raw	JSON	XML
8	200	HTTP	Tunnel to	www.higginbotham.com:443	3,612	Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.												
9	200	HTTPS	www.higginbotham.com	/	17,353	Secure Protocol: TLS 1.3												
10	200	HTTPS	www.googleapis.com	/oauth2/v4/token	399	Cipher Suite: TLS_AES_256_GCM_SHA384												
11	200	HTTPS	optimizationguide-p...	/v1:GetHints?	428	== Server Certificate ==												
12	200	HTTP	Tunnel to	safebrowsing.google.com...	10,052	[Version]												
13	200	HTTPS	safebrowsing.googl...	/safebrowsing/clientrepor...	32	V3												
14	200	HTTPS	www.higginbotham.com	/wp-includes/css/dist/bloc...	14,716	[Subject]												
15	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	35,238	CN=higginbotham.com												
16	200	HTTP	Tunnel to	cdnjs.cloudflare.com:443	2,748	Simple Name: higginbotham.com												
17	200	HTTP	Tunnel to	cdn.callrail.com:443	4,277	DNS Name: higginbotham.com												
18	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	6,980	[Issuer]												
19	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	23,260	CN=GTS CA 1P5, O=Google Trust Services LLC, C=US												
20	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	23,266	Simple Name: GTS CA 1P5												
21	200	HTTPS	www.higginbotham.com	/wp-content/themes/orbit...	26,907	DNS Name: GTS CA 1P5												
22	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	1,436													
23	200	HTTPS	www.higginbotham.com	/wp-includes/js/jquery/qu...	31,396													

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B BC 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 BC 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 BC AD B8 60 BF F2 BC 1E 43 34 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B BC 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 BC 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 BC AD B8 60 BF F2 BC 1E 43 34 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

[Thumbprint]
129157F7A5852AD23DF17FE2934E6396DB6B0251

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

[Public Key]
Algorithm: RSA
Length: 2048
Key Blob: 30 82 01 0a 02 82 01 01 00 ca b8 66 5f 8a 37 94 14 e1 2f e3 49 cf d7 ad cb 20 8c 54 93 6b af 66 56 02 11 26 db 20 13 00 a0 ce fe ae e9 ac 8d 68 8f a0 56 fd ff b4 b1 4b 9c 54 ff 55 4a 95 76 b9 db 3c 77 5e 9a b1 dd 30 74 36 77 cb 92 12 df 17 62 6d 2a aa 74 3c f6 68 c8 34 8d eb 69 97 9e f2 43 02 fa d7 76 99 c0 bc b3 56 2d 17 ca 39 c8 00 5d 91 c8 1d 9b c9 70 c8 c1 b0 da 40 23 3e 9a fe a9 ed e4 60 ce 15 b0 af 43 a3 07 a5 81 25 dd c3 92 e7 72 9e 27 df bd 01 05 2a b4 60 b8 83 6b fa bc db 97 51 47 a8 2e d0 52 7f c2 0a 1c 02 96 d2 c7 fd 04 a5 7e 78 04 ee 59 ed 33 9a e4 cc 42 bb 87 36 95 ac 4b 9b 6b 4f 96 4c ed 20 f7 bd 71 4b bc d8 91 81 3c be 56 c8 05 a6 e1 7f 66 85 88 46 91 96 a3 e4 e8 bb 45 f5 b8 78 25 4b 02 cb 23 e6 54 94 ed 94 f6 87 3d 29 55 31 eb 59 e9 a1 54 4c 50 b0 74 2c a7 b6 1e 7f 23 02 03 01 00 01
Parameters: 05 00

[Extensions]
* Key Usage(2.5.29.15):
Digital Signature, Key Encipherment (a0)

First decryption
algorithm

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm
00000019	63 6F 6D 3A 34 34 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 77									com:443 HTTP/1.1..Host: w
00000032	77 77 2E 68 69 67 67 69 6E 62 6F 74 68 61 6D 2E 63 6F 6D 3A 34 34 33 0D 0A									ww.migginbotham.com:443..
0000004B	43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 6B 65 65 70 2D 61 6C 69 76 65 0D 0A 55									Connection: keep-alive..U
00000064	73 65 72 2D 41 67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 57									ser-Agent: Mozilla/5.0 (W
0000007D	69 6E 64 6F 77 73 20 4E 54 20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36									indows NT10.0; Win64; x6
00000096	34 29 20 41 70 70 6C 65 57 65 62 4B 69 74 2F 35 33 37 2E 33 36 20 28 4B 48									4) AppleWebKit/537.36 (KH
000000AF	54 4D 4C 2C 20 6C 69 6B 65 20 47 65 63 6B 6F 29 20 43 68 72 6F 6D 65 2F 31									TML, like Gecko) Chrome/1
000000C8	32 36 2E 30 2E 30 2E 30 20 53 61 66 61 72 69 2F 35 33 37 2E 33 36 0D 0A 0D									26.0.0.0 Safari/537.36...
000000E1	0A 41 20 53 53 4C 76 33 2D 63 6F 6D 70 61 74 69 62 6C 65 20 43 6C 69 65 6E									.A SSLv3-compatible Clie
000000FA	74 48 65 6C 6C 6F 20 68 61 6E 64 73 68 61 6B 65 20 77 61 73 20 66 6F 75 6E									tHello handshake was foun
00000113	64 2E 20 46 69 64 64 6C 65 72 20 65 78 74 72 61 63 74 65 64 20 74 68 65 20									d. Fiddler extracted the
0000012C	70 61 72 61 6D 65 74 65 72 73 20 62 65 6C 6F 77 2E 0A 0A 53 65 63 75 72 65									parameters below...Secure
00000145	20 50 72 6F 74 6F 63 6F 6C 3A 20 54 4C 53 20 31 2E 33 0A 43 69 70 68 65 72									Protocol: TLS 1.3.Cipher
0000015E	20 53 75 69 74 65 3A 20 54 4C 53 5F 41 45 53 5F 32 35 36 5F 47 43 4D 5F 53									Suite: TLS_AES_256_GCM_S
00000177	48 41 33 38 34 0A 0A 52 65 63 6F 72 64 20 4C 61 79 65 72 20 56 65 72 73 69									HA384..Record Layer Versi
00000190	6F 6E 3A 20 33 2E 33 20 28 54 4C 53 2F 31 2E 32 29 0A 52 61 6E 64 6F 6D 3A									on: 3.3 (TLS/1.2).Random:
000001A9	20 46 36 20 39 44 20 31 45 20 30 35 20 33 44 20 35 38 20 35 33 20 35 30 20									F6 9D 1E 05 3D 58 53 50
000001C2	43 35 20 33 38 20 43 42 20 36 38 20 45 39 20 42 31 20 37 31 20 42 45 20 30									C5 38 CB 68 E9 B1 71 BE 0
000001DB	32 20 37 37 20 41 37 20 46 41 20 41 42 20 33 46 20 43 43 20 31 44 20 39 37									2 77 A7 FA AB 3F CC 1D 97
000001F4	20 31 42 20 34 43 20 41 46 20 41 42 20 37 41 20 43 44 20 36 39 0A 22 54 69									1B 4C AF AB 7A CD 69."Ti
0000020D	6D 65 22 3A 20 32 31 2D 30 39 2D 31 39 37 32 20 30 38 3A 33 33 3A 31 38 0A									me": 21-09-1972 08:33:18.
00000226	53 65 73 73 69 6F 6E 49 44 3A 20 35 45 20 42 33 20 34 42 20 37 30 20 31 32									SessionID: 5E B3 4B 70 12
0000023F	20 34 44 20 32 43 20 43 42 20 36 41 20 35 42 20 39 41 20 36 33 20 38 39 20									4D 2C CB 6A 5B 9A 63 89

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream
000004C9	36 37 20 39 38 20 39 46 20 38 42 20 38 38 20 44 39 20 41 30 20 43 36 20 31									67 98 9F 8B 88 D9 A0 C6 1
000004E2	35 20 43 43 20 33 37 20 42 42 20 35 31 20 39 31 20 31 44 20 33 30 20 43 45									5 CC 37 BB 51 91 1D 30 CE
000004FB	20 38 44 20 46 36 20 38 45 20 35 46 20 35 34 20 43 32 20 35 41 20 45 33 20									8D F6 8E 5F 54 C2 5A E3
00000514	38 39 20 32 43 20 37 42 20 38 39 20 41 41 20 42 42 20 43 35 20 41 46 20 45									39 2C 7B 89 AA BB C5 AF E
0000052D	39 20 35 36 20 31 30 20 42 38 20 31 35 20 35 37 20 39 34 20 41 31 20 41 31									9 56 10 B8 15 57 94 A1 A1
00000546	20 30 43 20 32 41 20 37 33 20 33 45 20 42 34 20 38 35 20 37 41 20 39 35 20									0C 2A 73 3E B4 85 7A 95
0000055F	34 34 20 33 34 20 30 37 20 31 39 20 43 38 20 33 33 20 33 46 20 45 43 20 39									44 34 07 19 C8 33 3F EC 9
00000578	43 20 34 35 20 34 36 20 30 42 20 38 34 20 43 39 20 31 43 20 32 32 20 45 31									C 45 46 0B 84 C9 1C 22 E1
00000591	20 33 43 20 39 46 20 38 37 20 37 31 20 41 33 20 35 42 20 43 33 20 37 32 20									3C 9F 87 71 A3 5B C3 72
000005AA	42 30 20 45 31 20 43 39 20 44 35 20 43 41 20 39 44 20 43 34 20 36 39 20 42									B0 E1 C9 D5 CA 9D C4 69 B
000005C3	30 20 30 35 20 35 42 20 39 44 20 45 32 20 41 34 20 32 42 20 35 36 20 46 35									0 05 5B 9D E2 A4 2B 56 F5
000005DC	20 42 46 20 38 32 20 38 34 20 36 37 20 39 43 20 43 36 20 36 35 20 35 46 20									BF 82 84 67 9C C6 65 5F
000005F5	38 32 20 33 45 20 42 32 20 46 30 20 42 42 20 46 35 20 33 42 20 41 46 20 30									82 3E B2 F0 BB F5 3B AF 0
0000060E	46 20 38 36 20 31 31 20 41 34 20 31 43 20 30 30 20 35 41 20 34 36 20 36 39									F 86 11 A4 1C 00 5A 46 69
00000627	20 44 39 20 35 37 20 41 46 20 33 31 20 43 36 20 43 32 20 46 34 20 46 30 20									D9 57 AF 31 C6 C2 F4 F0
00000640	33 38 20 46 37 20 45 43 20 39 37 20 36 41 20 32 31 20 37 38 20 43 34 20 41									38 F7 EC 97 6A 21 78 C4 A
00000659	38 20 41 41 20 42 34 20 38 35 20 43 45 20 43 39 20 38 35 20 36 37 20 44 37									8 AA B4 85 CE C9 85 67 D7
00000672	20 37 41 20 30 43 20 45 42 20 41 42 20 37 39 20 31 44 20 38 33 20 33 41 20									7A 0C EB AB 79 1D 83 3A
0000068B	42 32 20 33 42 20 36 30 20 44 36 20 41 33 20 43 32 20 30 31 20 38 37 20 41									B2 3B 60 D6 A3 C2 01 87 A
000006A4	37 20 46 37 20 43 43 20 42 38 20 41 30 20 45 35 20 31 45 20 31 46 20 35 35									7 F7 CC B8 A0 E5 1E 1F 55
000006BD	20 38 38 20 33 34 20 36 35 20 32 46 20 42 41 20 43 41 20 30 36 20 39 43 20									88 34 65 2F BA CA 06 9C
000006D6	33 42 20 31 37 20 39 44 20 41 36 20 31 31 20 42 32 20 33 39 20 39 46 20 33									3B 17 9D A6 11 B2 39 9F 3
000006EF	34 20 30 30 20 36 43 20 43 42 20 44 30 20 35 33 20 33 45 20 35 37 20 31 36									4 00 6C CB D0 53 3E 57 16

Source: Fiddler Capture

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3

Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate ==

[Version]
V3

[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com

[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US

Second decryption
algorithm

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type.

The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^{14} bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.2. Authenticated Decryption

Second decryption algorithm

The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block.

These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<-----> [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

	<p>The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A <i>cryptographic hash function</i> is a function that computes a <i>message authentication code</i> from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.</p> <p>https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf</p>
21. The system of claim 20, wherein the decrypting is done using a key associated with each decryption algorithm.	The standard practices the method such that the decrypting is done using a key (e.g., decryption key) associated with each decryption algorithm (e.g., signature decryption algorithm such as SHA-256RSA, etc., and AEAD decryption algorithm such as TLS_AES_256_GCM_SHA384, etc.).

The Higginbotham Difference

About Us

Careers

Blog

Locations

Become a Higginbotham Partner

Report a Claim

H Higginbotham™

Business Insurance

Employee Benefits

HR Services

Financial Services

Personal Insurance

Captive Insurance

Let's Talk

Insurance Brokers Who are Principled, Perceptive and Personal

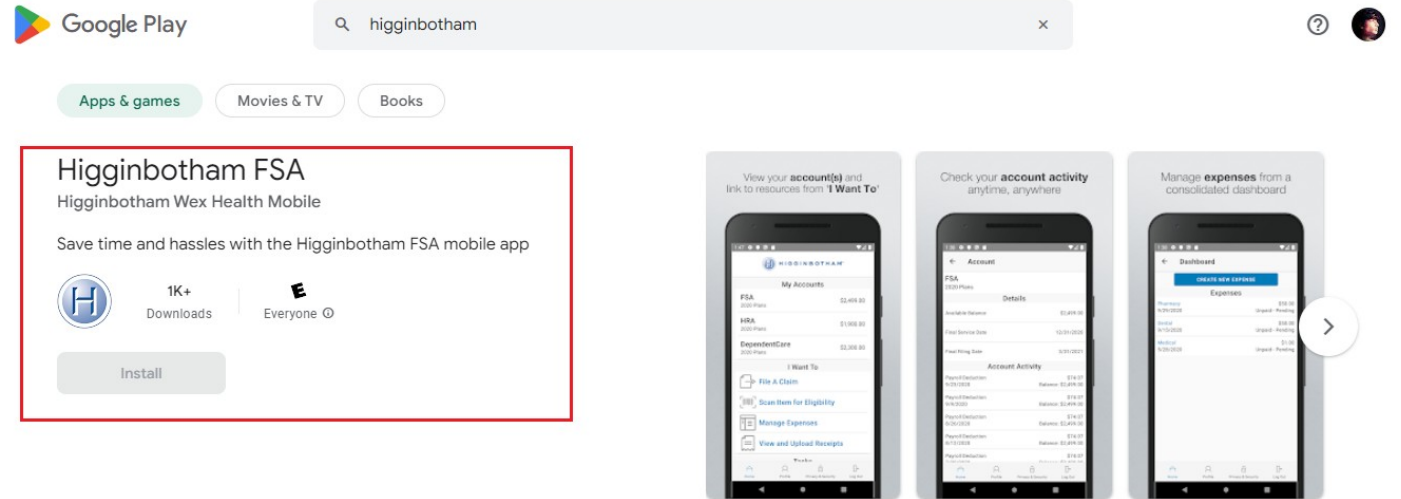
A single source solution inspired by your needs.

Request a Quote →

Let's Talk →

PAUSE VIDEO

<https://www.higginbotham.com/>



<https://play.google.com/store/search?q=higginbotham&c=apps&hl=en&gl=US>

```
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
  extended_master_secret  empty
  ec_point_formats  uncompressed [0x0]
  status_request  OCSP - Implicit Responder
  psk_key_exchange_modes  01 01
  renegotiation_info  00
  0xfe0d  00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 DD 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
  server_name  www.higginbotham.com
  csease(0x3a3a)  00
```

Source: Fiddler Capture

As shown below, the signature decryption algorithm utilizes a private key for a first decryption and the AEAD decryption algorithm uses a key K. Both the decryption techniques are decrypting using their respective associated keys.

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p><u>2.2. Authenticated Decryption</u></p> <p>The <u>authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).</u></p> <p>https://datatracker.ietf.org/doc/html/rfc5116</p> <p>The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. <u>The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.</u></p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
22. The system of claim 21, wherein the key is resident in hardware of the target unit or the key is retrieved from a server.	The standard utilized by the accused instrumentality practices the method such that the key is resident in hardware (e.g., stored in a memory storage of the server such as a database, RAM, etc.) of the target unit (e.g., server of the accused instrumentality) or the key is retrieved from a server.

The Higginbotham Difference About Us Careers Blog Locations Become a Higginbotham Partner Search Report a Claim

Higginbotham Business Insurance Employee Benefits HR Services Financial Services Personal Insurance Captive Insurance Let's Talk

Insurance Brokers Who are Principled, Perceptive and Personal

A single source solution inspired by your needs.

Request a Quote -> Let's Talk ->

PAUSE VIDEO

<https://www.higginbotham.com/>



```
JA 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
    extended_master_secret    empty
    ec_point_formats    uncompressed [0x0]
    status_request    OCSP - Implicit Responder
    psk_key_exchange_modes    01 01
    renegotiation_info    00
    00 00 01 00 01 8B 00 20 08 F8 C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 04 5E 00 D0 A7 B0 A8 9E C4 E1 F7 BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D0 29 05 15
    0xfeed
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 88 BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C A8 B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
    server_name    www.higginbotham.com
    orease (0x3a3a)    00
```

Source: Fiddler Capture



Tech Accelerator

Server hardware guide: Architecture, products and management

3. Random access memory



RAM is the main type of memory in a computing system.

RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>



Tech Accelerator

Server hardware guide: Architecture, products and management

f

X

in



4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

<https://www.techtargget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>

As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

<https://datatracker.ietf.org/doc/html/rfc8446#>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2.2. Authenticated Decryption

The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

23. The system of claim 22, wherein the key is contained in a key data structure.

The standard utilized by the accused instrumentality practices the method such that the key (e.g., private key, Key K, etc.) is contained in a key data structure (e.g., data structure).

The Higginbotham Difference About Us Careers Blog Locations Become a Higginbotham Partner Search Report a Claim

Higginbotham Business Insurance Employee Benefits HR Services Financial Services Personal Insurance Captive Insurance Let's Talk

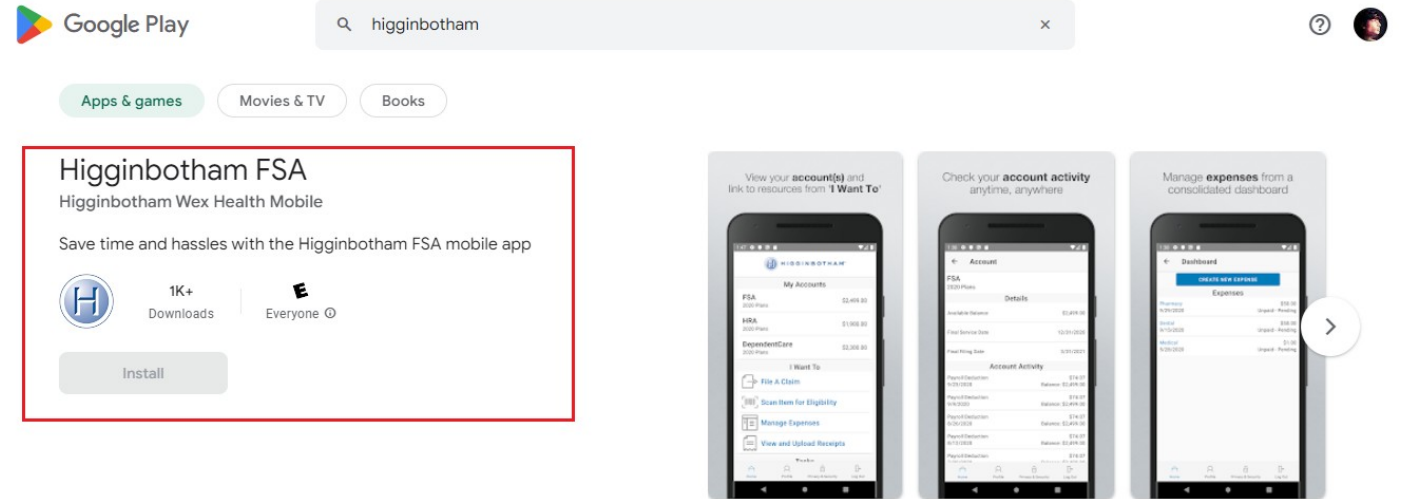
Insurance Brokers Who are Principled, Perceptive and Personal

A single source solution inspired by your needs.

Request a Quote -> Let's Talk ->

PAUSE VIDEO

<https://www.higginbotham.com/>



<https://play.google.com/store/search?q=higginbotham&c=apps&hl=en&gl=US>

```
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 DD 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
orease(0x3a3a) 00
```

Source: Fiddler Capture

The accused instrumentality utilizes a server to establish a secure TLS communication with a client. The server must comprise a memory storage and store data according to a data structure to implement the standard efficiently.



Tech Accelerator

Server hardware guide: Architecture, products and management

3. Random access memory



RAM is the main type of memory in a computing system.

RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as

data to be moved to a storage device. Thus, RAM works very

closely with the processor and must match the processor's

incredible speed and performance. This kind of fast memory

is usually termed dynamic RAM, and several DRAM

variations are available for servers.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>



Tech Accelerator

Server hardware guide: Architecture, products and management



4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>

A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it.

In computer science and computer programming, a data structure may be selected or designed to store data for the purpose of using it with various algorithms. In some cases, the algorithm's basic operations are tightly coupled to the data structure's design. Each data structure contains information about the data values, relationships between the data and -- in some cases -- functions that can be applied to the data.

<https://www.techtarget.com/searchdatamanagement/definition/data-structure>

As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

	<p>Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. <u>This is necessary for the ClientHello storage mechanism described in Section 8.2</u> because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.</p> <p>https://datatracker.ietf.org/doc/html/rfc8446#</p>
--	--

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p><u>2.2. Authenticated Decryption</u></p> <p>The <u>authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).</u></p> <p>https://datatracker.ietf.org/doc/html/rfc5116</p> <p>The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. <u>The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.</u></p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>29. The system of claim 21, wherein each encryption algorithm is a symmetric key system or an asymmetric key system.</p>	<p>The standard practices the method such that each encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA, etc., and AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.) is a symmetric key system (e.g., AEAD encryption algorithm, etc.) or an asymmetric key system (e.g., signature encryption algorithm).</p> <p>As shown below, the server comprises a memory storage to store messages for</p>

establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. The standard defines the signature encryption algorithm as an asymmetric cryptography algorithm and the AEAD encryption algorithm as the symmetric cryptography algorithm.

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

<https://datatracker.ietf.org/doc/html/rfc8446#>

Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK).

<https://datatracker.ietf.org/doc/html/rfc8446#section-4>

cipher_suites: A list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. Values are defined in [Appendix B.4](#). If the list contains cipher suites that the server does not recognize, support, or wish to use, the server MUST ignore those cipher suites and process the remaining ones as usual. If the client is attempting a PSK key establishment, it SHOULD advertise at least one cipher suite indicating a Hash associated with the PSK.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

	<p><u>2.2. Authenticated Decryption</u></p> <p>The <u>authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).</u></p> <p>https://datatracker.ietf.org/doc/html/rfc5116</p> <p>The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. <u>The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.</u></p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>30. The system of claim 21, further operable for associating a first Message Authentication Code (MAC) or first digital signature with each</p>	<p>The standard practices associating a first Message Authentication Code (MAC) (e.g., message authentication code with hashing function) or first digital signature with each encrypted bit stream (e.g., encrypted bit stream with the signature encryption algorithm i.e., SHA256RSA, etc., and encrypted bitstream with the AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.).</p> <p>As shown below, the standard discloses a hashing function with each of the encryption</p>

encrypted bit stream.

algorithm. It performs a message authentication code with the utilized hashing function.

```
SignedCertTimestamp (RFC6962)  empty
ALPN  h2, http/1.1
signature_algs  ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b  02 00 02
supported_groups  grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share  04 ED 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 1D 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 B0 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F3 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 06 E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B BC 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F5
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 FB 9F E9 09 14 5A E3 B6 5A 5B 9B D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD D0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 84 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 6E 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 4B 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 BC 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 A2 E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 6B 38 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 BC AD B8 60 BF F2 8C 1E 43 34 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 A6 D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret  empty
ec_point_formats  uncompressed [0x0]
status_request  OCSF - Implicit Responder
psk_key_exchange_modes  01 01
renegotiation_info  00
0xfe0d  00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 82 C5 69 CC 12 85 91 94 13 32 DD 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name  www.higginbotham.com
grease (0x3a3a)  00
```

Source: Fiddler Capture

First encryption algorithm

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm																									
00000019	63	6F	6D	3A	34	34	33	20	48	54	54	50	2F	31	2E	31	0D	0A	48	6F	73	74	3A	20	77	com:443 HTTP/1.1..Host: w									
00000032	77	77	2E	68	69	67	67	69	6E	62	6F	74	68	61	6D	2E	63	6F	6D	3A	34	34	33	0D	0A	ww.higginbotham.com:443..									
0000004B	43	6F	6E	6E	65	63	74	69	6F	6E	3A	20	6B	65	65	70	2D	61	6C	69	76	65	0D	0A	55	Connection: keep-alive..U									
00000064	73	65	72	2D	41	67	65	6E	74	3A	20	4D	6F	7A	69	6C	6E	61	2F	35	2E	30	20	28	57	ser-Agent: Mozilla/5.0 (W									
0000007D	69	6E	64	6F	77	73	20	4E	54	20	31	30	2E	30	3B	20	57	69	6E	36	34	3B	20	78	36	indows NT10.0; Win64; x6									
00000096	34	29	20	41	70	70	6C	65	57	65	62	4B	69	74	2F	35	33	37	2E	33	36	20	28	4B	48	4) AppleWebKit/537.36 (KH									
000000AF	54	4D	4C	2C	20	6C	69	6B	65	20	47	65	63	6B	6F	29	20	43	68	72	6F	6D	65	2F	31	TML, like Gecko) Chrome/1									
000000C8	32	36	2E	30	2E	30	2E	30	20	53	61	66	61	72	69	2F	35	33	37	2E	33	36	0D	0A	0D	26.0.0.0 Safari/537.36...									
000000E1	0A	41	20	53	53	4C	76	33	2D	63	6F	6D	70	61	74	69	62	6C	65	20	43	6C	69	65	6E	.A SSLv3-compatible Clie									
000000FA	74	48	65	6C	6C	6F	20	68	61	6E	64	73	68	61	6B	65	20	77	61	73	20	66	6F	75	6E	tHello handshake was foun									
00000113	64	2E	20	46	69	64	64	6C	65	72	20	65	78	74	72	61	63	74	65	64	20	74	68	65	20	d. Fiddler extracted the									
0000012C	70	61	72	61	6D	65	74	65	72	73	20	62	65	6C	6F	77	2E	0A	0A	53	65	63	75	72	65	parameters below...Secure									
00000145	20	50	72	6F	74	6F	63	6F	6C	3A	20	54	4C	53	20	31	2E	33	0A	43	69	70	68	65	72	Protocol: TLS 1.3.Cipher									
0000015E	20	53	75	69	74	65	3A	20	54	4C	53	5F	41	45	53	5F	32	35	36	5F	47	43	4D	5F	53	Suite: TLS AES_256_GCM_S									
00000177	48	41	33	38	34	0A	0A	52	65	63	6F	72	64	20	4C	61	79	65	72	20	56	65	72	73	69	HA384..Record Layer Versi									
00000190	6F	6E	3A	20	33	2E	33	20	28	54	4C	53	2F	31	2E	32	29	0A	52	61	6E	64	6F	6D	3A	on: 3.3 (TLS/1.2).Random:									
000001A9	20	46	36	20	39	44	20	31	45	20	30	35	20	33	44	20	35	38	20	35	33	20	35	30	20	F6 9D 1E 05 3D 58 53 50									
000001C2	43	35	20	33	38	20	43	42	20	36	38	20	45	39	20	42	31	20	37	31	20	42	45	20	30	C5 38 A7 68 E9 B1 71 BE 0									
000001DB	32	20	37	37	20	41	37	20	46	41	20	41	42	20	33	46	20	43	43																

Second encryption algorithm

```
com:443 HTTP/1.1..Host: w
ww.higginbotham.com:443..
Connection: keep-alive..U
ser-Agent: Mozilla/5.0 (W
indows NT 10.0; Win64; x6
4) AppleWebKit/537.36 (KH
TML, like Gecko) Chrome/1
26.0.0.0 Safari/537.36...
..A SSLv3-compatible Clien
tHello handshake was foun
d. Fiddler extracted the
parameters below...Secure
Protocol: TLS 1.3.Cipher
Suite: TLS_AES_256_GCM_S
HA384..Record Layer Versi
on: 3.3 (TLS/1.2).Random:
f6 9D 1E 05 3D 58 53 50
C5 38 FB 68 E9 B1 71 BE 0
2 77 A7 FA AB 3F CC 1D 97
1B 4C AF AB 7A CD 69."Ti
me": 21:09-1972 08:33:18.
SessionID: 5E B3 4B 70 12
4D 2C CB 6A 5B 9A 63 89
```


Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream
000004C9	36 37 20 39 38 20 39 46 20 38 42 20 38 38 20 44 39 20 41 30 20 43 36 20 31									67 98 9F 8B 88 D9 A0 C6 1
000004E2	35 20 43 43 20 33 37 20 42 42 20 35 31 20 39 31 20 31 44 20 33 30 20 43 45									5 CC 37 BB 51 91 1D 30 CE
000004FB	20 38 44 20 46 36 20 38 45 20 35 46 20 35 34 20 43 32 20 35 41 20 45 33 20									8D F6 8E 5F 54 C2 5A E3
00000514	38 39 20 32 43 20 37 42 20 38 39 20 41 41 20 42 42 20 43 35 20 41 46 20 45									39 2C 7B 89 AA BB C5 AF E
0000052D	39 20 35 36 20 31 30 20 42 38 20 31 35 20 35 37 20 39 34 20 41 31 20 41 31									9 56 10 B8 15 57 94 A1 A1
00000546	20 30 43 20 32 41 20 37 33 20 33 45 20 42 34 20 38 35 20 37 41 20 39 35 20									0C 2A 73 3E B4 85 7A 95
0000055F	34 34 20 33 34 20 30 37 20 31 39 20 43 38 20 33 33 20 33 46 20 45 43 20 39									44 34 07 19 C8 33 3F EC 9
00000578	43 20 34 35 20 34 36 20 30 42 20 38 34 20 43 39 20 31 43 20 32 32 20 45 31									C 45 46 0B 84 C9 1C 22 E1
00000591	20 33 43 20 39 46 20 38 37 20 37 31 20 41 33 20 35 42 20 43 33 20 37 32 20									3C 9F 87 71 A3 5B C3 72
000005AA	42 30 20 45 31 20 43 39 20 44 35 20 43 41 20 39 44 20 43 34 20 36 39 20 42									B0 E1 C9 D5 CA 9D C4 69 B
000005C3	30 20 30 35 20 35 42 20 39 44 20 45 32 20 41 34 20 32 42 20 35 36 20 46 35									0 05 5B 9D E2 A4 2B 56 F5
000005DC	20 42 46 20 38 32 20 38 34 20 36 37 20 39 43 20 43 36 20 36 35 20 35 46 20									BF 82 84 67 9C C6 65 5F
000005F5	38 32 20 33 45 20 42 32 20 46 30 20 42 42 20 46 35 20 33 42 20 41 46 20 30									82 3E B2 F0 BB F5 3B AF 0
0000060E	46 20 38 36 20 31 31 20 41 34 20 31 43 20 30 30 20 35 41 20 34 36 20 36 39									F 86 11 A4 1C 00 5A 46 69
00000627	20 44 39 20 35 37 20 41 46 20 33 31 20 43 36 20 43 32 20 46 34 20 46 30 20									D9 57 AF 31 C6 C2 F4 F0
00000640	33 38 20 46 37 20 45 43 20 39 37 20 36 41 20 32 31 20 37 38 20 43 34 20 41									38 F7 EC 97 6A 21 78 C4 A
00000659	38 20 41 41 20 42 34 20 38 35 20 43 45 20 43 39 20 38 35 20 36 37 20 44 37									8 AA B4 85 CE C9 85 67 D7
00000672	20 37 41 20 30 43 20 45 42 20 41 42 20 37 39 20 31 44 20 38 33 20 33 41 20									7A 0C EB AB 79 1D 83 3A
0000068B	42 32 20 33 42 20 36 30 20 44 36 20 41 33 20 43 32 20 30 31 20 38 37 20 41									B2 3B 60 D6 A3 C2 01 87 A
000006A4	37 20 46 37 20 43 43 20 42 38 20 41 30 20 45 35 20 31 45 20 31 46 20 35 35									7 F7 CC B8 A0 E5 1E 1F 55
000006BD	20 38 38 20 33 34 20 36 35 20 32 46 20 42 41 20 43 41 20 30 36 20 39 43 20									88 34 65 2F BA CA 06 9C
000006D6	33 42 20 31 37 20 39 44 20 41 36 20 31 31 20 42 32 20 33 39 20 39 46 20 33									3B 17 9D A6 11 B2 39 9F 3
000006EF	34 20 30 30 20 36 43 20 43 42 20 44 30 20 35 33 20 33 45 20 35 37 20 31 36									4 00 6C CB D0 53 3E 57 16

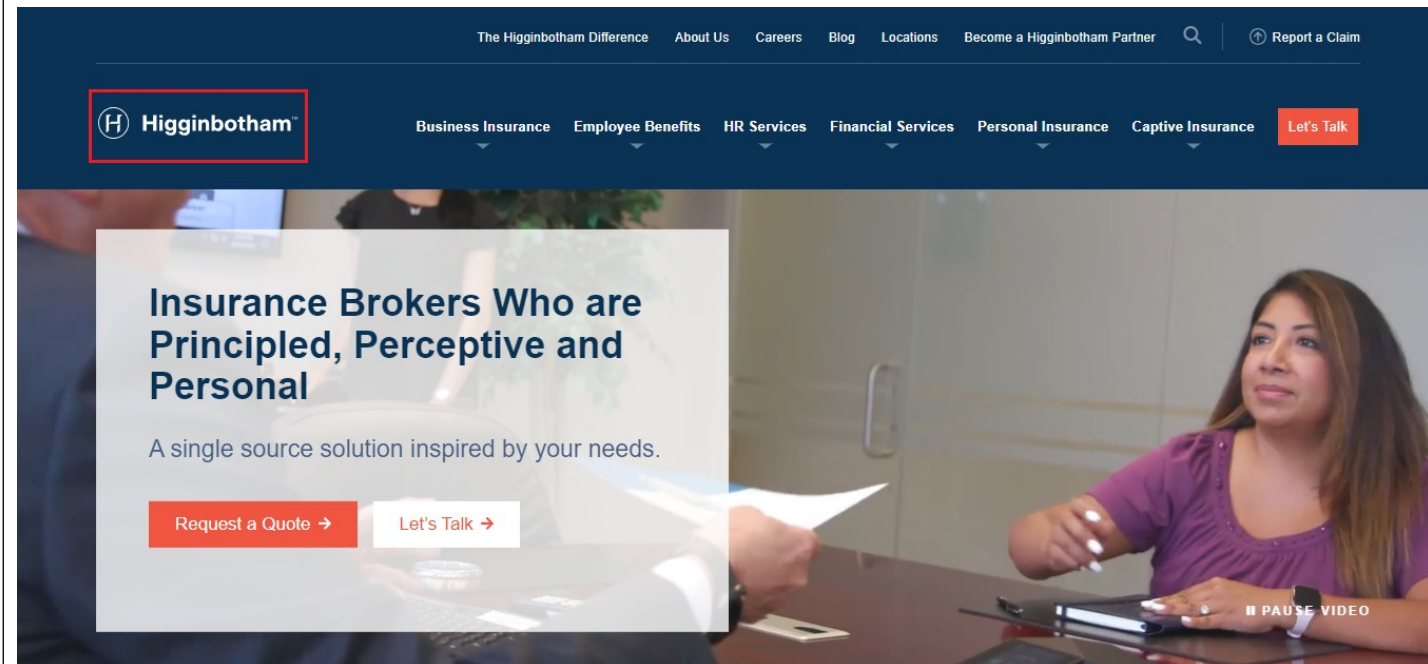
Source: Fiddler Capture

	<p>The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A <i>cryptographic hash function</i> is a function that computes a <i>message authentication code</i> from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.</p> <p>https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf</p> <p>The list of supported symmetric encryption algorithms has been pruned of all algorithms that are considered legacy. Those that remain are all Authenticated Encryption with Associated Data (AEAD) algorithms. The cipher suite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with both the key derivation function and <u>handshake message authentication code (MAC)</u>.</p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-4</p>
37. A computer storage device for a recursive	The accused instrumentality utilizes a computer storage device (e.g., a memory of the server of the accused instrumentality) for a recursive security protocol (e.g., TLS 1.3


security protocol for protecting digital content, comprising instructions executable by a processor for performing the steps of:

security protocol) for protecting digital content (e.g., digital certificate related to the accused instrumentality), comprising instructions executable by a processor (e.g., a processor of the server of the accused instrumentality).

The accused instrumentality utilizes TLS 1.3 security protocol (hereinafter “the standard”) for communicating content such as digital certificate, application data, etc., with a client. The standard provides a two-level encryption security. It encrypts a plaintext with a first encryption technique and generates a ciphertext. Further, it encrypts the ciphertext with a second encryption technique i.e., recursive encryption security.



<https://www.higginbotham.com/>



Google Play

Apps & games


Movies & TV

Books

Higginbotham FSA


Higginbotham Wex Health Mobile

Save time and hassles with the Higginbotham FSA mobile app



1K+

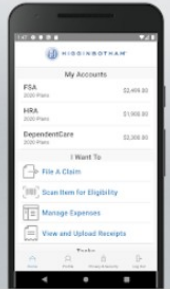
Downloads



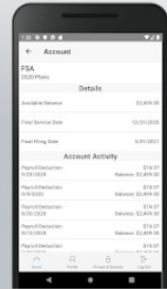
Everyone 0

Install

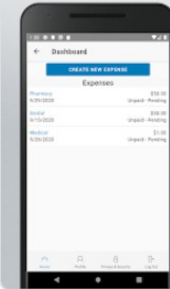
View your **account(s)** and link to resources from **I Want To**



Check your **account activity** anytime, anywhere



Manage **expenses** from a consolidated dashboard



<https://play.google.com/store/search?q=higginbotham&c=apps&hl=en&gl=US>

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

As shown below, the accused instrumentality utilizes a two-level algorithm security. It utilizes the SHA256RSA encryption algorithm as a first encryption algorithm i.e., signature encryption algorithm and the TLS_AES_256_GCM_SHA384 encryption algorithm as a second encryption algorithm i.e., AEAD encryption algorithm.

8200HTTPTunnel towww.higginbotham.com:4433,612

9200HTTPSwww.higginbotham.com/17,353

10200HTTPSwww.googleapis.com/oauth2/v4/token399

11200HTTPSoptimizationguide-p.../v1:GetHints?428

12200HTTPTunnel tosafebrowsing.google.com...10,052

13200HTTPSsafebrowsing.google.com/safebrowsing/clientrepor...32

14200HTTPSwww.higginbotham.com/wp-includes/css/dist/bloc...14,716

15200HTTPSwww.higginbotham.com/wp-content/cache/min/1/...35,238

16200HTTPTunnel tocdnjs.cloudflare.com:4432,748

17200HTTPTunnel tocdn.callrail.com:4434,277

18200HTTPSwww.higginbotham.com/wp-content/cache/min/1/...6,980

19200HTTPSwww.higginbotham.com/wp-content/cache/min/1/...23,260

20200HTTPSwww.higginbotham.com/wp-content/cache/min/1/...23,266

21200HTTPSwww.higginbotham.com/wp-content/themes/orbit...26,907

22200HTTPSwww.higginbotham.com/wp-content/cache/min/1/...1,436

23200HTTPSwww.higginbotham.com/wp-includes/js/jquery/qu...31,396

TransformerHeadersTextViewSyntaxViewImageViewHexViewWebViewAuthCachingCookiesRawJSONXML

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3
Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate ==
[Version]
V3
[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com
[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US
Simple Name: GTS CA 1P5
DNS Name: GTS CA 1P5

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 D9 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 C8 B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B 8C 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD D0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 8C 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 8C AD B8 60 BF F2 BC 1E 43 24 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 D9 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 C8 B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B 8C 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD D0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 8C 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 8C AD B8 60 BF F2 BC 1E 43 24 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

[Thumbprint]
129157F7A5852AD23DF17FE2934E6396DB6B0251

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption
algorithm

[Public Key]

Algorithm: RSA
Length: 2048

Key Blob: 30 82 01 0a 02 82 01 01 00 ca b8 66 5f 8a 37 94 14 e1 2f e3 49 cf d7 ad cb 20 8c 54 93 6b af 66 56 02 11 26 db 20 13 00 a0 ce fe ae e9 ac 8d 68 8f a0 56 fd ff b4 b1 4b 9c 54 ff 55 4a 95 76 b9 db 3c 77 5e 9a b1 dd 30 74 36 77 cb 92 12 df 17 62 6d 2a aa 74 3c f6 68 c8 34 8d eb 69 97 9e f2 43 02 fa d7 76 99 c0 bc b3 56 2d 17 ca 39 c8 00 5d 91 c8 1d 9b c9 70 c8 c1 b0 da 40 23 3e 9a fe a9 ed e4 60 ce 15 b0 af 43 a3 07 a5 81 25 dd c3 92 e7 72 9e 27 df bd 01 05 2a b4 60 b8 83 6b fa bc db 97 51 47 a8 2e d0 52 7f c2 0a 1c 02 96 d2 c7 fd 04 a5 7e 78 04 ee 59 ed 33 9a e4 cc 42 bb 87 36 95 ac 4b 9b 6b 4f 96 4c ed 20 f7 bd 71 4b bc d8 91 81 3c be 56 c8 05 a6 e1 7f 66 85 88 46 91 96 a3 e4 e8 bb 45 f5 b8 78 25 4b 02 cb 23 e6 54 94 ed 94 f6 87 3d 29 55 31 eb 59 e9 a1 54 4c 50 b0 74 2c a7 b6 1e 7f 23 02 03 01 00 01

Parameters: 05 00

[Extensions]

* Key Usage(2.5.29.15):
Digital Signature, Key Encipherment (a0)

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm
00000019	63 6F 6D 3A 34 34 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 77									com:443 HTTP/1.1..Host: w
00000032	77 77 2E 68 69 67 67 69 6E 62 6F 74 68 61 6D 2E 63 6F 6D 3A 34 34 33 0D 0A									ww.migginbotham.com:443..
0000004B	43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 6B 65 65 70 2D 61 6C 69 76 65 0D 0A 55									Connection: keep-alive..U
00000064	73 65 72 2D 41 67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 57									ser-Agent: Mozilla/5.0 (W
0000007D	69 6E 64 6F 77 73 20 4E 54 20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36									indows NT10.0; Win64; x6
00000096	34 29 20 41 70 70 6C 65 57 65 62 4B 69 74 2F 35 33 37 2E 33 36 20 28 4B 48									4) AppleWebKit/537.36 (KH
000000AF	54 4D 4C 2C 20 6C 69 6B 65 20 47 65 63 6B 6F 29 20 43 68 72 6F 6D 65 2F 31									TML, like Gecko) Chrome/1
000000C8	32 36 2E 30 2E 30 2E 30 20 53 61 66 61 72 69 2F 35 33 37 2E 33 36 0D 0A 0D									26.0.0.0 Safari/537.36...
000000E1	0A 41 20 53 53 4C 76 33 2D 63 6F 6D 70 61 74 69 62 6C 65 20 43 6C 69 65 6E									.A SSLv3-compatible Clie
000000FA	74 48 65 6C 6C 6F 20 68 61 6E 64 73 68 61 6B 65 20 77 61 73 20 66 6F 75 6E									tHello handshake was foun
00000113	64 2E 20 46 69 64 64 6C 65 72 20 65 78 74 72 61 63 74 65 64 20 74 68 65 20									d. Fiddler extracted the
0000012C	70 61 72 61 6D 65 74 65 72 73 20 62 65 6C 6F 77 2E 0A 0A 53 65 63 75 72 65									parameters below...Secure
00000145	20 50 72 6F 74 6F 63 6F 6C 3A 20 54 4C 53 20 31 2E 33 0A 43 69 70 68 65 72									Protocol: TLS 1.3.Cipher
0000015E	20 53 75 69 74 65 3A 20 54 4C 53 5F 41 45 53 5F 32 35 36 5F 47 43 4D 5F 53									Suite: TLS_AES_256_GCM_S
00000177	48 41 33 38 34 0A 0A 52 65 63 6F 72 64 20 4C 61 79 65 72 20 56 65 72 73 69									HA384..Record Layer Versi
00000190	6F 6E 3A 20 33 2E 33 20 28 54 4C 53 2F 31 2E 32 29 0A 52 61 6E 64 6F 6D 3A									on: 3.3 (TLS/1.2).Random:
000001A9	20 46 36 20 39 44 20 31 45 20 30 35 20 33 44 20 35 38 20 35 33 20 35 30 20									F6 9D 1E 05 3D 58 53 50
000001C2	43 35 20 33 38 20 43 42 20 36 38 20 45 39 20 42 31 20 37 31 20 42 45 20 30									C5 38 CB 68 E9 B1 71 BE 0
000001DB	32 20 37 37 20 41 37 20 46 41 20 41 42 20 33 46 20 43 43 20 31 44 20 39 37									2 77 A7 FA AB 3F CC 1D 97
000001F4	20 31 42 20 34 43 20 41 46 20 41 42 20 37 41 20 43 44 20 36 39 0A 22 54 69									1B 4C AF AB 7A CD 69."Ti
0000020D	6D 65 22 3A 20 32 31 2D 30 39 2D 31 39 37 32 20 30 38 3A 33 33 3A 31 38 0A									me": 21-09-1972 08:33:18.
00000226	53 65 73 73 69 6F 6E 49 44 3A 20 35 45 20 42 33 20 34 42 20 37 30 20 31 32									SessionID: 5E B3 4B 70 12
0000023F	20 34 44 20 32 43 20 43 42 20 36 41 20 35 42 20 39 41 20 36 33 20 38 39 20									4D 2C CB 6A 5B 9A 63 89

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream																								
000004C9	36	37	20	39	38	20	39	46	20	38	42	20	38	38	20	44	39	20	41	30	20	43	36	20	31	67	98	9F	8B	88	D9	A0	C6	1
000004E2	35	20	43	43	20	33	37	20	42	42	20	35	31	20	39	31	20	31	44	20	33	30	20	43	45	5	CC	37	BB	51	91	1D	30	CE
000004FB	20	38	44	20	46	36	20	38	45	20	35	46	20	35	34	20	43	32	20	35	41	20	45	33	20	8D	F6	8E	5F	54	C2	5A	E3	
00000514	38	39	20	32	43	20	37	42	20	38	39	20	41	41	20	42	42	20	43	35	20	41	46	20	45	39	2C	7B	89	AA	BB	C5	AF	E
0000052D	39	20	35	36	20	31	30	20	42	38	20	31	35	20	35	37	20	39	34	20	41	31	20	41	31	9	56	10	B8	15	57	94	A1	A1
00000546	20	30	43	20	32	41	20	37	33	20	33	45	20	42	34	20	38	35	20	37	41	20	39	35	20	0C	2A	73	3E	B4	85	7A	95	
0000055F	34	34	20	33	34	20	30	37	20	31	39	20	43	38	20	33	33	20	33	46	20	45	43	20	39	44	34	07	19	C8	33	3F	EC	9
00000578	43	20	34	35	20	34	36	20	30	42	20	38	34	20	43	39	20	31	43	20	32	32	20	45	31	C	45	46	0B	84	C9	1C	22	E1
00000591	20	33	43	20	39	46	20	38	37	20	37	31	20	41	33	20	35	42	20	43	33	20	37	32	20	3C	9F	87	71	A3	5B	C3	72	
000005AA	42	30	20	45	31	20	43	39	20	44	35	20	43	41	20	39	44	20	43	34	20	36	39	20	42	B0	E1	C9	D5	CA	9D	C4	69	B
000005C3	30	20	30	35	20	35	42	20	39	44	20	45	32	20	41	34	20	32	42	20	35	36	20	46	35	0	05	5B	9D	E2	A4	2B	56	F5
000005DC	20	42	46	20	38	32	20	38	34	20	36	37	20	39	43	20	43	36	20	36	35	20	35	46	20	BF	82	84	67	9C	C6	65	5F	
000005F5	38	32	20	33	45	20	42	32	20	46	30	20	42	42	20	46	35	20	33	42	20	41	46	20	30	82	3E	B2	F0	BB	F5	3B	AF	0
0000060E	46	20	38	36	20	31	31	20	41	34	20	31	43	20	30	30	20	35	41	20	34	36	20	36	39	F	86	11	A4	1C	00	5A	46	69
00000627	20	44	39	20	35	37	20	41	46	20	33	31	20	43	36	20	43	32	20	46	34	20	46	30	20	D9	57	AF	31	C6	C2	F4	F0	
00000640	33	38	20	46	37	20	45	43	20	39	37	20	36	41	20	32	31	20	37	38	20	43	34	20	41	38	F7	EC	97	6A	21	78	C4	A
00000659	38	20	41	41	20	42	34	20	38	35	20	43	45	20	43	39	20	38	35	20	36	37	20	44	37	8	AA	B4	85	CE	C9	85	67	D7
00000672	20	37	41	20	30	43	20	45	42	20	41	42	20	37	39	20	31	44	20	38	33	20	33	41	20	7A	0C	EB	AB	79	1D	83	3A	
0000068B	42	32	20	33	42	20	36	30	20	44	36	20	41	33	20	43	32	20	30	31	20	38	37	20	41	B2	3B	60	D6	A3	C2	01	87	A
000006A4	37	20	46	37	20	43	43	20	42	38	20	41	30	20	45	35	20	31	45	20	31	46	20	35	35	7	F7	CC	B8	A0	E5	1E	1F	55
000006BD	20	38	38	20	33	34	20	36	35	20	32	46	20	42	41	20	43	41	20	30	36	20	39	43	20	88	34	65	2F	BA	CA	06	9C	
000006D6	33	42	20	31	37	20	39	44	20	41	36	20	31	31	20	42	32	20	33	39	20	39	46	20	33	3B	17	9D	A6	11	B2	39	9F	3
000006EF	34	20	30	30	20	36	43	20	43	42	20	44	30	20	35	33	20	33	45	20	35	37	20	31	36	4	00	6C	CB	D0	53	3E	57	16

Source: Fiddler Capture

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3

Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate ==

[Version]

V3

[Subject]

CN=higginbotham.com

Simple Name: higginbotham.com

DNS Name: higginbotham.com

[Issuer]

CN=GTS CA 1P5, O=Google Trust Services LLC, C=US

Second decryption
algorithm

Source: Fiddler Capture

As shown below, the server of the accused instrumentality comprises a processor to

execute instructions and a memory storage to store instructions for performing the operations defined by the standard.

```
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret  empty
ec_point_formats  uncompressed [0x0]
status_request  OCSP - Implicit Responder
psk_key_exchange_modes  01 01
renegotiation_info  00
0xfe0d  00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 DD 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 88 BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 85 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name  www.higginbotham.com
crease (0x3a3a)  00
```

Source: Fiddler Capture



Tech Accelerator

Server hardware guide: Architecture, products and management



2. Processor

The CPU -- or simply [processor](#) -- is a complex micro-circuitry device that serves as the foundation of all computer operations. It supports hundreds of possible commands hardwired into hundreds of millions of transistors to process low-level software instructions -- microcode -- and data and derive a desired logical or mathematical result. The processor works closely with memory, which both holds the software instructions and data to be processed as well as the results or output of those processor operations.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>



Tech Accelerator

Server hardware guide: Architecture, products and management

3. Random access memory



RAM is the main type of memory in a computing system.

RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>



Tech Accelerator

Server hardware guide: Architecture, products and management



4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

<https://www.techtargget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

<https://datatracker.ietf.org/doc/html/rfc8446#>

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

Client

Server

Key ^ ClientHello

Exch | + key_share*

```
| + signature_algorithms*
```

```
| + psk_key_exchange_modes*
```

$$v + \text{pre_shared_key}^* \rightarrow$$

ServerHello ^ Key

key_share*	Exch
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14
15	15
16	16
17	17
18	18
19	19
20	20
21	21
22	22
23	23
24	24
25	25
26	26
27	27
28	28
29	29
30	30
31	31
32	32
33	33
34	34
35	35
36	36
37	37
38	38
39	39
40	40
41	41
42	42
43	43
44	44
45	45
46	46
47	47
48	48
49	49
50	50
51	51
52	52
53	53
54	54
55	55
56	56
57	57
58	58
59	59
60	60
61	61
62	62
63	63
64	64
65	65
66	66
67	67
68	68
69	69
70	70
71	71
72	72
73	73
74	74
75	75
76	76
77	77
78	78
79	79
80	80
81	81
82	82
83	83
84	84
85	85
86	86
87	87
88	88
89	89
90	90
91	91
92	92
93	93
94	94
95	95
96	96
97	97
98	98
99	99
100	100

$$+ \text{pre_shared_key}^*$$

```
{EncryptedExtensions}
```

```
{CertificateRequest*}
```

```
{Certificate*}
```

```
{CertificateVerify*}
```

```
{Finished}
```

```
<----- [Application Data*]
```

Digital Content

$$\wedge \{ \text{Certificate}^* \}$$

Auth | {CertificateVerify*}

v {Finished}

[Application Data]

----->

```
<-----> [Application Data]
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

Second
encryption

- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.

First
encryption

- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.
- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Introduction

The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties:

- Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated.

Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK).

- Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques.
- Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection.

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2¹⁴ bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [\[RFC5116\], Section 4](#)). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>This specification defines the following cipher suites for use with TLS 1.3.</p> <table border="1"> <thead> <tr> <th>Description</th><th>Value</th></tr> </thead> <tbody> <tr> <td>TLS_AES_128_GCM_SHA256</td><td>{0x13,0x01}</td></tr> <tr> <td>TLS_AES_256_GCM_SHA384</td><td>{0x13,0x02}</td></tr> <tr> <td>TLS_CHACHA20_POLY1305_SHA256</td><td>{0x13,0x03}</td></tr> <tr> <td>TLS_AES_128_CCM_SHA256</td><td>{0x13,0x04}</td></tr> <tr> <td>TLS_AES_128_CCM_8_SHA256</td><td>{0x13,0x05}</td></tr> </tbody> </table> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>	Description	Value	TLS_AES_128_GCM_SHA256	{0x13,0x01}	TLS_AES_256_GCM_SHA384	{0x13,0x02}	TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}	TLS_AES_128_CCM_SHA256	{0x13,0x04}	TLS_AES_128_CCM_8_SHA256	{0x13,0x05}
Description	Value												
TLS_AES_128_GCM_SHA256	{0x13,0x01}												
TLS_AES_256_GCM_SHA384	{0x13,0x02}												
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}												
TLS_AES_128_CCM_SHA256	{0x13,0x04}												
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}												
encrypting a bit stream with a first encryption algorithm;	<p>The standard practices encrypting a bitstream (e.g., bitstream of digital certificate) with a first encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA encryption algorithm).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext.</p>												

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

As shown below, the accused instrumentality discloses the signature encryption algorithm.

8200HTTPSTunnel towww.higginbotham.com:4433,612

9200HTTPSwww.higginbotham.com/17,353

10200HTTPSwww.googleapis.com/oauth2/v4/token399

11200HTTPSoptimizationguide-p.../v1:GetHints?428

12200HTTPTunnel tosafebrowsing.google.com...10,052

13200HTTPSsafebrowsing.google.com/safebrowsing/clientrepor...32

14200HTTPSwww.higginbotham.com/wp-includes/css/dist/bloc...14,716

15200HTTPSwww.higginbotham.com/wp-content/cache/min/1/...35,238

16200HTTPTunnel tocdnjs.cloudflare.com:4432,748

17200HTTPTunnel tocdn.callrail.com:4434,277

18200HTTPSwww.higginbotham.com/wp-content/cache/min/1/...6,980

19200HTTPSwww.higginbotham.com/wp-content/cache/min/1/...23,260

20200HTTPSwww.higginbotham.com/wp-content/cache/min/1/...23,266

21200HTTPSwww.higginbotham.com/wp-content/themes/orbit...26,907

22200HTTPSwww.higginbotham.com/wp-content/cache/min/1/...1,436

23200HTTPSwww.higginbotham.com/wp-includes/js/jquery/jqu...31,396

TransformerHeadersTextViewSyntaxViewImageViewHexViewWebViewAuthCachingCookiesRawJSONXML

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3
Cipher Suite: TLS_AES_256_GCM_SHA384

-- Server Certificate =====
[Version]
V3

[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com

[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US
Simple Name: GTS CA 1P5
DNS Name: GTS CA 1P5

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B BC 0A B5 2C 21 80 CA 1F E2 AC BA 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 BC 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 BC AD B8 60 BF F2 BC 1E 43 34 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B BC 0A B5 2C 21 80 CA 1F E2 AC BA 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD 0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 BC 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 BC AD B8 60 BF F2 BC 1E 43 34 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
     | {CertificateVerify*}
     v {Finished}
       [Application Data] ----->
<----- [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.
- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.
- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A First encryption "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.
- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>Introduction</p> <p>The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream. Specifically, the secure channel should provide the following properties:</p> <ul style="list-style-type: none"> - Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032]) or a symmetric pre-shared key (PSK). - Confidentiality: Data sent over the channel after establishment is only visible to the endpoints. TLS does not hide the length of the data it transmits, though endpoints are able to pad TLS records in order to obscure lengths and improve protection against traffic analysis techniques. - Integrity: Data sent over the channel after establishment cannot be modified by attackers without detection. <p>https://datatracker.ietf.org/doc/html/rfc8446</p>
<p>associating a first decryption algorithm with the encrypted bit stream;</p>	<p>The standard practices associating a first decryption algorithm (e.g., signature decryption algorithm i.e., SHA256RSA decryption algorithm) with the encrypted bit stream (e.g., encrypted certificate with signature encryption algorithm).</p> <p>The standard practices providing a two-level encryption security for data</p>

communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA encryption algorithm) and generates a ciphertext.

The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate.

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

As shown below, the accused instrumentality discloses the signature decryption algorithm.

[Thumbprint]
129157F7A5852AD23DF17FE2934E6396DB6B0251

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption
algorithm

[Public Key]

Algorithm: RSA
Length: 2048

Key Blob: 30 82 01 0a 02 82 01 01 00 ca b8 66 5f 8a 37 94 14 e1 2f e3 49 cf d7 ad cb 20 8c 54 93 6b af 66 56 02 11 26 db 20 13 00 a0 ce fe ae e9 ac 8d 68 8f a0 56 fd ff b4 b1 4b 9c 54 ff 55 4a 95 76 b9 db 3c 77 5e 9a b1 dd 30 74 36 77 cb 92 12 df 17 62 6d 2a aa 74 3c f6 68 c8 34 8d eb 69 97 9e f2 43 02 fa d7 76 99 c0 bc b3 56 2d 17 ca 39 c8 00 5d 91 c8 1d 9b c9 70 c8 c1 b0 da 40 23 3e 9a fe a9 ed e4 60 ce 15 b0 af 43 a3 07 a5 81 25 dd c3 92 e7 72 9e 27 df bd 01 05 2a b4 60 b8 83 6b fa bc db 97 51 47 a8 2e d0 52 7f c2 0a 1c 02 96 d2 c7 fd 04 a5 7e 78 04 ee 59 ed 33 9a e4 cc 42 bb 87 36 95 ac 4b 9b 6b 4f 96 4c ed 20 f7 bd 71 4b bc d8 91 81 3c be 56 c8 05 a6 e1 7f 66 85 88 46 91 96 a3 e4 e8 bb 45 f5 b8 78 25 4b 02 cb 23 e6 54 94 ed 94 f6 87 3d 29 55 31 eb 59 e9 a1 54 4c 50 b0 74 2c a7 b6 1e 7f 23 02 03 01 00 01

Parameters: 05 00

[Extensions]

* Key Usage(2.5.29.15):

Digital Signature, Key Encipherment (a0)

Source: Fiddler Capture

OID description

First decryption algorithm identifier

OID:	{iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-1(1) sha256WithRSAEncryption(11)}	(ASN.1 notation)
	1.2.840.113549.1.1.11	(dot notation)
	/ISO/Member-Body/US/113549/1/1/11	(OID-IRI notation)

Description:

Public-Key Cryptography Standards (PKCS) #1 version 1.5 signature algorithm with Secure Hash Algorithm 256 (SHA256) and Rivest, Shamir and Adleman (RSA) encryption

<http://oid-info.com/get/1.2.840.113549.1.1.11>

-- When the following OIDs are used in an AlgorithmIdentifier, the
-- parameters MUST be present and MUST be NULL.

sha224WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 14 }

sha256WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 11 }

sha384WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 12 }

sha512WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 13 }

<https://www.ietf.org/rfc/rfc4055.txt>

Figure 1 below shows the basic full TLS handshake:

Client

Server

Key ^ ClientHello

Exch | + key_share*

```
| + signature_algorithms*
```

```
| + psk_key_exchange_modes*
```

$$v + \text{pre_shared_key}^* \rightarrow$$

ServerHello ^ Key

+ key_share* | Exch

$$+ \text{pre_shared_key}^*$$

```
{EncryptedExtensions}
```

```
{CertificateRequest*}
```

```
{Certificate*}
```

```
{CertificateVerify*} | Auth
```

```
{Finished}
```

```
<----- [Application Data*]
```

Digital Content

$$\wedge \{ \text{Certificate}^* \}$$

Auth | {CertificateVerify*}

```
v {Finished}
```

[Application Data]

----->

```
<-----> [Application Data]
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<----- [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.
- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.
- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A First encryption "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.
- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

First
decryption
algorithm
information

The algorithm field specifies the signature algorithm used (see [Section 4.2.3](#) for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in [Section 4.4.1](#), namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$




First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

	<p>The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A <i>cryptographic hash function</i> is a function that computes a <i>message authentication code</i> from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.</p> <p>https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf</p>
<p>encrypting both the encrypted bit stream and the first decryption algorithm with a second encryption algorithm to yield a second bit stream;</p>	<p>The standard practices encrypting both the encrypted bit stream (e.g., encrypted digital certificate) and the first decryption algorithm (e.g., signature decryption algorithm) with a second encryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) to yield a second bit stream (e.g., TLS ciphertext bitstream).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</p> <p>The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it.</p>

The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.

Security overview

This page is secure (valid HTTPS).

- Certificate - valid and trusted
The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.
[View certificate](#)
- Connection - secure connection settings
The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.
- Resources - all served securely
All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm
00000019	63 6F 6D 3A 34 34 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 77									con:443 HTTP/1.1..Host: w
00000032	77 77 2E 68 69 67 67 69 6E 62 6F 74 68 61 6D 2E 63 6F 6D 3A 34 34 33 0D 0A									ww.higginbotham.com:443..
0000004B	43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 6B 65 65 70 2D 61 6C 69 76 65 0D 0A 55									Connection: keep-alive..U
00000064	73 65 72 2D 41 67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 57									ser-Agent: Mozilla/5.0 (W
0000007D	69 6E 64 6F 77 73 20 4E 54 20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36									indows NT 10.0; Win64; x6
00000096	34 29 20 41 70 70 6C 65 57 65 62 4B 69 74 2F 35 33 37 2E 33 36 20 28 4B 48									4) AppleWebKit/537.36 (KH
000000AF	54 4D 4C 2C 20 6C 69 6B 65 20 47 65 63 6B 6F 29 20 43 68 72 6F 6D 65 2F 31									TML, like Gecko) Chrome/1
000000C8	32 36 2E 30 2E 30 2E 30 20 53 61 66 61 72 69 2F 35 33 37 2E 33 36 0D 0A 0D									26.0.0.0 Safari/537.36...
000000E1	0A 41 20 53 53 4C 76 33 2D 63 6F 6D 70 61 74 69 62 6C 65 20 43 6C 69 65 6E									.A SSLv3-compatible Clie
000000FA	74 48 65 6C 6C 6F 20 68 61 6E 64 73 68 61 6B 65 20 77 61 73 20 66 6F 75 6E									tHello handshake was foun
00000113	64 2E 20 46 69 64 64 6C 65 72 20 65 78 74 72 61 63 74 65 64 20 74 68 65 20									d. Fiddler extracted the
0000012C	70 61 72 61 6D 65 74 65 72 73 20 62 65 6C 6F 77 2E 0A 0A 53 65 63 75 72 65									parameters below...Secure
00000145	20 50 72 6F 74 6F 63 6F 6C 3A 20 54 4C 53 20 31 2E 33 0A 43 69 70 68 65 72									Protocol: TLS 1.3.Cipher
0000015E	20 53 75 69 74 65 3A 20 54 4C 53 5F 41 45 53 5F 32 35 36 5F 47 43 4D 5F 53									Suite: TLS_AES_256_GCM_S
00000177	48 41 33 38 34 0A 0A 52 65 63 6F 72 64 20 4C 61 79 65 72 20 56 65 72 73 69									HA384..Record Layer Versi
00000190	6F 6E 3A 20 33 2E 33 20 28 54 4C 53 2F 31 2E 32 29 0A 52 61 6E 64 6F 6D 3A									on: 3.3 (TLS/1.2).Random:
000001A9	20 46 36 20 39 44 20 31 45 20 30 35 20 33 44 20 35 38 20 35 33 20 35 30 20									F6 9D 1E 05 3D 58 53 50
000001C2	43 35 20 33 38 20 43 42 20 36 38 20 45 39 20 42 31 20 37 31 20 42 45 20 30									C5 38 CB 68 E9 B1 71 BE 0
000001DB	32 20 37 37 20 41 37 20 46 41 20 41 42 20 33 46 20 43 43 20 31 44 20 39 37									2 77 A7 FA AB 3F CC 1D 97
000001F4	20 31 42 20 34 43 20 41 46 20 41 42 20 37 41 20 43 44 20 36 39 0A 22 54 69									1B 4C AF AB 7A CD 69."Ti
0000020D	6D 65 22 3A 20 32 31 2D 30 39 2D 31 39 37 32 20 30 38 3A 33 33 3A 31 38 0A									me": 21-09-1972 08:33:18.
00000226	53 65 73 73 69 6F 6E 49 44 3A 20 35 45 20 42 33 20 34 42 20 37 30 20 31 32									SessionID: 5E B3 4B 70 12
0000023F	20 34 44 20 32 43 20 43 42 20 36 41 20 35 42 20 39 41 20 36 33 20 38 39 20									4D 2C CB 6A 5B 9A 63 89

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream
000004C9	36 37 20 39 38 20 39 46 20 38 42 20 38 38 20 44 39 20 41 30 20 43 36 20 31									67 98 9F 8B 88 D9 A0 C6 1
000004E2	35 20 43 43 20 33 37 20 42 42 20 35 31 20 39 31 20 31 44 20 33 30 20 43 45									5 CC 37 BB 51 91 1D 30 CE
000004FB	20 38 44 20 46 36 20 38 45 20 35 46 20 35 34 20 43 32 20 35 41 20 45 33 20									8D F6 8E 5F 54 C2 5A E3
00000514	38 39 20 32 43 20 37 42 20 38 39 20 41 41 20 42 42 20 43 35 20 41 46 20 45									9 2C 7B 89 AA BB C5 AF E
0000052D	39 20 35 36 20 31 30 20 42 38 20 31 35 20 35 37 20 39 34 20 41 31 20 41 31									9 56 10 B8 15 57 94 A1 A1
00000546	20 30 43 20 32 41 20 37 33 20 33 45 20 42 34 20 38 35 20 37 41 20 39 35 20									0C 2A 73 3E B4 85 7A 95
0000055F	34 34 20 33 34 20 30 37 20 31 39 20 43 38 20 33 33 20 33 46 20 45 43 20 39									44 34 07 19 C8 33 3F EC 9
00000578	43 20 34 35 20 34 36 20 30 42 20 38 34 20 43 39 20 31 43 20 32 32 20 45 31									C 45 46 0B 84 C9 1C 22 E1
00000591	20 33 43 20 39 46 20 38 37 20 37 31 20 41 33 20 35 42 20 43 33 20 37 32 20									3C 9F 87 71 A3 5B C3 72
000005AA	42 30 20 45 31 20 43 39 20 44 35 20 43 41 20 39 44 20 43 34 20 36 39 20 42									B0 E1 C9 D5 CA 9D C4 69 B
000005C3	30 20 30 35 20 35 42 20 39 44 20 45 32 20 41 34 20 32 42 20 35 36 20 46 35									0 05 5B 9D E2 A4 2B 56 F5
000005DC	20 42 46 20 38 32 20 38 34 20 36 37 20 39 43 20 43 36 20 36 35 20 35 46 20									BF 82 84 67 9C C6 65 5F
000005F5	38 32 20 33 45 20 42 32 20 46 30 20 42 42 20 46 35 20 33 42 20 41 46 20 30									82 3E B2 F0 BB F5 3B AF 0
0000060E	46 20 38 36 20 31 31 20 41 34 20 31 43 20 30 30 20 35 41 20 34 36 20 36 39									F 86 11 A4 1C 00 5A 46 69
00000627	20 44 39 20 35 37 20 41 46 20 33 31 20 43 36 20 43 32 20 46 34 20 46 30 20									D9 57 AF 31 C6 C2 F4 F0
00000640	33 38 20 46 37 20 45 43 20 39 37 20 36 41 20 32 31 20 37 38 20 43 34 20 41									38 F7 EC 97 6A 21 78 C4 A
00000659	38 20 41 41 20 42 34 20 38 35 20 43 45 20 43 39 20 38 35 20 36 37 20 44 37									8 AA B4 85 CE C9 85 67 D7
00000672	20 37 41 20 30 43 20 45 42 20 41 42 20 37 39 20 31 44 20 38 33 20 33 41 20									7A 0C EB AB 79 1D 83 3A
0000068B	42 32 20 33 42 20 36 30 20 44 36 20 41 33 20 43 32 20 30 31 20 38 37 20 41									B2 3B 60 D6 A3 C2 01 87 A
000006A4	37 20 46 37 20 43 43 20 42 38 20 41 30 20 45 35 20 31 45 20 31 46 20 35 35									7 F7 CC B8 A0 E5 1E 1F 55
000006BD	20 38 38 20 33 34 20 36 35 20 32 46 20 42 41 20 43 41 20 30 36 20 39 43 20									88 34 65 2F BA CA 06 9C
000006D6	33 42 20 31 37 20 39 44 20 41 36 20 31 31 20 42 32 20 33 39 20 39 46 20 33									3B 17 9D A6 11 B2 39 9F 3
000006EF	34 20 30 30 20 36 43 20 43 42 20 44 30 20 35 33 20 33 45 20 35 37 20 31 36									4 00 6C CB D0 53 3E 57 16

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext

handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2¹⁴ bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding `TLSPlaintext.length` due to the inclusion of `TLSInnerPlaintext.type` and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (`iv_length`) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [\[RFC5116\], Section 4](#)). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

All handshake messages after the ServerHello are now encrypted.
The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block. These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<-----> [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.1.1. Cryptographic Negotiation

In TLS, the cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello:

- A list of cipher suites which indicates the AEAD algorithm/HKDF hash pairs which the client supports.

Second
encryption

- A "supported_groups" ([Section 4.2.7](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.8](#)) extension which contains (EC)DHE shares for some or all of these groups.

First
encryption

- A "signature_algorithms" ([Section 4.2.3](#)) extension which indicates the signature algorithms which the client can accept. A "signature_algorithms_cert" extension ([Section 4.2.3](#)) may also be added to indicate certificate-specific signature algorithms.
- A "pre_shared_key" ([Section 4.2.11](#)) extension which contains a list of symmetric key identities known to the client and a "psk_key_exchange_modes" ([Section 4.2.9](#)) extension which indicates the key exchange modes that may be used with PSKs.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

First
decryption
algorithm
information

The algorithm field specifies the signature algorithm used (see [Section 4.2.3](#) for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in [Section 4.4.1](#), namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>The "<u>extension_data</u>" field of these extensions contains a <u>SignatureSchemeList</u> value:</p> <pre> enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), </pre> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>associating a second decryption algorithm with the second bit stream.</p>	<p>The standard practices associating a second decryption algorithm (e.g., cipher suit selected from one of the AEAD algorithms such as TLS_AES_256_GCM_SHA384, etc.) with the second bit stream (e.g., TLS ciphertext bitstream).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</p>

The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

					TransformerHeadersTextViewSyntaxViewImageViewHexViewWebViewAuthCachingCookiesRawJSONXML									
8	200	HTTP	Tunnel to www.higginbotham.com:443	3,612										
9	200	HTTPS	www.higginbotham.com /	17,353										
10	200	HTTPS	www.googleapis.com /oauth2/v4/token	399										
11	200	HTTPS	optimizationguide-p... /v1:GetHints?	428										
12	200	HTTP	Tunnel to safebrowsing.google.com...	10,052										
13	200	HTTPS	safebrowsing.google... /safebrowsing/clientrepor...	32										
14	200	HTTPS	www.higginbotham.com /wp-includes/css/dist/bloc...	14,716										
15	200	HTTPS	www.higginbotham.com /wp-content/cache/min/1/...	35,238										
16	200	HTTP	Tunnel to cdnjs.cloudflare.com:443	2,748										
17	200	HTTP	Tunnel to cdn.cloudflare.com:443	4,277										
18	200	HTTPS	www.higginbotham.com /wp-content/cache/min/1/...	6,980										
19	200	HTTPS	www.higginbotham.com /wp-content/cache/min/1/...	23,260										
20	200	HTTPS	www.higginbotham.com /wp-content/cache/min/1/...	23,266										
21	200	HTTPS	www.higginbotham.com /wp-content/themes/orbit...	26,907										
22	200	HTTPS	www.higginbotham.com /wp-content/cache/min/1/...	1,436										
23	200	HTTPS	www.higginbotham.com /wp-includes/js/jquery/jqu...	31,396										

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3

Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate ==

[Version]V3

[Subject]CN=higginbotham.comSimple Name: higginbotham.comDNS Name: higginbotham.com

[Issuer]CN=GTS CA 1P5, O=Google Trust Services LLC, C=USSimple Name: GTS CA 1P5DNS Name: GTS CA 1P5

Source: Fiddler Capture

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3

Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate =====
[Version]
V3

Second decryption
algorithm

[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com

[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream
000004C9	36 37 20 39 38 20 39 46 20 38 42 20 38 38 20 44 39 20 41 30 20 43 36 20 31									67 98 9F 8B 88 D9 A0 C6 1
000004E2	35 20 43 43 20 33 37 20 42 42 20 35 31 20 39 31 20 31 44 20 33 30 20 43 45									5 CC 37 BB 51 91 1D 30 CE
000004FB	20 38 44 20 46 36 20 38 45 20 35 46 20 35 34 20 43 32 20 35 41 20 45 33 20									8D F6 8E 5F 54 C2 5A E3
00000514	38 39 20 32 43 20 37 42 20 38 39 20 41 41 20 42 42 20 43 35 20 41 46 20 45									39 2C 7B 89 AA BB C5 AF E
0000052D	39 20 35 36 20 31 30 20 42 38 20 31 35 20 35 37 20 39 34 20 41 31 20 41 31									9 56 10 B8 15 57 94 A1 A1
00000546	20 30 43 20 32 41 20 37 33 20 33 45 20 42 34 20 38 35 20 37 41 20 39 35 20									0C 2A 73 3E B4 85 7A 95
0000055F	34 34 20 33 34 20 30 37 20 31 39 20 43 38 20 33 33 20 33 46 20 45 43 20 39									44 34 07 19 C8 33 3F EC 9
00000578	43 20 34 35 20 34 36 20 30 42 20 38 34 20 43 39 20 31 43 20 32 32 20 45 31									C 45 46 0B 84 C9 1C 22 E1
00000591	20 33 43 20 39 46 20 38 37 20 37 31 20 41 33 20 35 42 20 43 33 20 37 32 20									3C 9F 87 71 A3 5B C3 72
000005AA	42 30 20 45 31 20 43 39 20 44 35 20 43 41 20 39 44 20 43 34 20 36 39 20 42									B0 E1 C9 D5 CA 9D C4 69 B
000005C3	30 20 30 35 20 35 42 20 39 44 20 45 32 20 41 34 20 32 42 20 35 36 20 46 35									0 05 5B 9D E2 A4 2B 56 F5
000005DC	20 42 46 20 38 32 20 38 34 20 36 37 20 39 43 20 43 36 20 36 35 20 35 46 20									BF 82 84 67 9C C6 65 5F
000005F5	38 32 20 33 45 20 42 32 20 46 30 20 42 42 20 46 35 20 33 42 20 41 46 20 30									82 3E B2 F0 BB F5 3B AF 0
0000060E	46 20 38 36 20 31 31 20 41 34 20 31 43 20 30 30 20 35 41 20 34 36 20 36 39									F 86 11 A4 1C 00 5A 46 69
00000627	20 44 39 20 35 37 20 41 46 20 33 31 20 43 36 20 43 32 20 46 34 20 46 30 20									D9 57 AF 31 C6 C2 F4 F0
00000640	33 38 20 46 37 20 45 43 20 39 37 20 36 41 20 32 31 20 37 38 20 43 34 20 41									38 F7 EC 97 6A 21 78 C4 A
00000659	38 20 41 41 20 42 34 20 38 35 20 43 45 20 43 39 20 38 35 20 36 37 20 44 37									8 AA B4 85 CE C9 85 67 D7
00000672	20 37 41 20 30 43 20 45 42 20 41 42 20 37 39 20 31 44 20 38 33 20 33 41 20									7A 0C EB AB 79 1D 83 3A
0000068B	42 32 20 33 42 20 36 30 20 44 36 20 41 33 20 43 32 20 30 31 20 38 37 20 41									B2 3B 60 D6 A3 C2 01 87 A
000006A4	37 20 46 37 20 43 43 20 42 38 20 41 30 20 45 35 20 31 45 20 31 46 20 35 35									7 F7 CC B8 A0 E5 1E 1F 55
000006BD	20 38 38 20 33 34 20 36 35 20 32 46 20 42 41 20 43 41 20 30 36 20 39 43 20									88 34 65 2F BA CA 06 9C
000006D6	33 42 20 31 37 20 39 44 20 41 36 20 31 31 20 42 32 20 33 39 20 39 46 20 33									3B 17 9D A6 11 B2 39 9F 3
000006EF	34 20 30 30 20 36 43 20 43 42 20 44 30 20 35 33 20 33 45 20 35 37 20 31 36									4 00 6C CB D0 53 3E 57 16

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type. The handshake messages are communicated to establish a secure channel for TLS

communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^{14} bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.2. Authenticated Decryption

Second decryption algorithm

The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [\[RFC5116\], Section 4](#)). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block.

These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data] ----->
<-----> [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p>The <u>"extension_data" field of these extensions contains a SignatureSchemeList value:</u></p> <pre> enum { /* RSASSA-PKCS1-v1_5 algorithms */ rsa_pkcs1_sha256(0x0401), rsa_pkcs1_sha384(0x0501), rsa_pkcs1_sha512(0x0601), /* ECDSA algorithms */ ecdsa_secp256r1_sha256(0x0403), ecdsa_secp384r1_sha384(0x0503), ecdsa_secp521r1_sha512(0x0603), /* RSASSA-PSS algorithms with public key OID rsaEncryption */ rsa_pss_rsae_sha256(0x0804), rsa_pss_rsae_sha384(0x0805), rsa_pss_rsae_sha512(0x0806), /* EdDSA algorithms */ ed25519(0x0807), ed448(0x0808), /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */ rsa_pss_pss_sha256(0x0809), rsa_pss_pss_sha384(0x080a), rsa_pss_pss_sha512(0x080b), </pre> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
38. The software system or computer program of claim 37, further translatable for	<p>The standard further discloses decrypting the first bit stream (e.g., encrypted digital certificate with signature encryption algorithm i.e., SHA-256 RSA, etc.) and the second bit stream (e.g., a second-level encryption with AEAD encryption algorithm such as TLS AES 256 GCM SHA384, etc.) with the first associated decryption</p>

<p>decrypting the first bit stream and the second bit stream with the first associated decryption algorithm and the second associated decryption algorithm wherein the decryption is accomplished by a target unit.</p>	<p>algorithm (e.g., signature decryption algorithm i.e., SHA-256 RSA, etc.) and the second associated decryption algorithm (e.g., cipher suit selected from one of the AEAD decryption algorithms such as TLS_AES_256_GCM_SHA384, etc.) wherein the decryption is accomplished by a target unit (e.g., a server of the accused instrumentality).</p> <p>The standard practices providing a two-level encryption security for data communication. It encrypts a plaintext with a first encryption technique i.e., signature encryption algorithm (e.g., SHA256RSA algorithm) and generates a ciphertext.</p> <p>The standard defines an authentication message, communicated after the hello handshake messages, which comprises an encrypted digital certificate with the signature encryption algorithm and an associated certificate verify message with it. The certificate verify message includes a signature algorithm extension field which provides information for the decryption of the encrypted digital certificate. The standard further practices encrypting the authentication message, including the encrypted digital certification and the certificate verify message, with a second decryption algorithm i.e., AEAD algorithm such as TLS_AES_256_GCM_SHA384, etc.</p>
---	---

Security overview



This page is secure (valid HTTPS).

■ Certificate - valid and trusted

The connection to this site is using a valid, trusted server certificate issued by GTS CA 1P5.

[View certificate](#)

■ Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, X25519Kyber768Draft00, and AES_128_GCM.

■ Resources - all served securely

All resources on this page are served securely.

<https://www.higginbotham.com/>

The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

<https://datatracker.ietf.org/doc/html/rfc8446>

					TransformerHeadersTextViewSyntaxViewImageViewHexViewWebViewAuthCachingCookiesRawJSONXML											
8	200	HTTP	Tunnel to	www.higginbotham.com:443	3,612	Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.										
9	200	HTTPS	www.higginbotham.com	/	17,353	Secure Protocol: TLS 1.3										
10	200	HTTPS	www.googleapis.com	/oauth2/v4/token	399	Cipher Suite: TLS_AES_256_GCM_SHA384										
11	200	HTTPS	optimizationguide-p...	/v1:GetHints?	428	== Server Certificate ==										
12	200	HTTP	Tunnel to	safebrowsing.google.com...	10,052	[Version]										
13	200	HTTPS	safebrowsing.google...	/safebrowsing/clientrepor...	32	V3										
14	200	HTTPS	www.higginbotham.com	/wp-includes/css/dist/bloc...	14,716	[Subject]										
15	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	35,238	CN=higginbotham.com										
16	200	HTTP	Tunnel to	cdnjs.cloudflare.com:443	2,748	Simple Name: higginbotham.com										
17	200	HTTP	Tunnel to	cdn.callrail.com:443	4,277	DNS Name: higginbotham.com										
18	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	6,980	[Issuer]										
19	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	23,260	CN=GTS CA 1P5, O=Google Trust Services LLC, C=US										
20	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	23,266	Simple Name: GTS CA 1P5										
21	200	HTTPS	www.higginbotham.com	/wp-content/themes/orbit...	26,907	DNS Name: GTS CA 1P5										
22	200	HTTPS	www.higginbotham.com	/wp-content/cache/min/1/...	1,436											
23	200	HTTPS	www.higginbotham.com	/wp-includes/js/jquery/qu...	31,396											

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B BC 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD D0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 BC 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 BC AD B8 60 BF F2 BC 1E 43 34 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

```
SignedCertTimestamp(RFC6962) empty
ALPN h2, http/1.1
signature_algs ecdsa_secp256r1_sha256,rsa_pss_rsae_sha256,rsa_pkcs1_sha256,ecdsa_secp384r1_sha384,rsa_pss_rsae_sha384,rsa_pkcs1_sha384,rsa_pss_rsae_sha512,rsa_pkcs1_sha512
0x001b 02 00 02
supported_groups grease [0x5a5a], unknown [0x6399], x25519 [0x1d], secp256r1 [0x17], secp384r1 [0x18]
key_share 04 ED 5A 5A 00 01 00 63 99 04 C0 73 FD 09 6A 32 61 2F 32 5D 09 69 67 98 9F 8B 88 D9 A0 C6 15 CC 37 BB 51 91 D0 30 CE 8D F6 8E 5F 54 C2 5A E3 89 2C 7B 89 AA BB C5 AF E9 56 10 B8 15 57 94 A1 A1 0C 2A 73 3E B4 85
7A 95 44 34 07 19 C8 33 3F EC 9C 45 46 0B 84 C9 1C 22 E1 3C 9F 87 71 A3 5B C3 72 80 E1 C9 D5 CA 9D C4 69 B0 05 58 9D E2 A4 2B 56 F5 BF 82 84 67 9C C6 65 5F 82 3E B2 F0 BB F5 3B AF 0F 86 11 A4 1C 00 5A 46 69 D9 57 AF 31 C6 C2 F4 F0 38 F7
EC 97 6A 21 78 C4 A8 AA B4 85 CE C9 85 67 D7 7A 0C EB AB 79 1D 83 3A B2 3B 60 D6 A3 C2 01 87 A7 F7 CC B8 A0 E5 1E 1F 55 88 34 65 2F BA CA 06 9C 3B 17 9D A6 11 B2 39 9F 34 00 6C CB D0 53 3E 57 16 0E E2 72 B7 D8 42 36 EA 03 9B DB 8E 2F
C0 61 1D A0 54 55 73 A1 99 23 8C 39 BA A0 C6 73 7A 95 49 8A 6B 1A 69 74 99 49 55 5B BC 0A B5 2C 21 80 CA 1F E2 AC B4 73 A5 FD EB 7B 6D B5 93 D6 10 37 A1 63 C8 18 C6 49 D3 71 0F E0 75 BA 39 AA 0B 66 56 BD DC 3C 1E 5D 8A 7E 1A B6 2E F7
23 C1 C3 B1 5E E8 05 13 61 7C 42 DC 1C 3F F0 AB 47 A3 00 28 E6 A7 C1 C1 26 7B D7 BF 9F E9 09 14 5A C3 B6 5A 5B 98 D2 D5 47 0D 6B 7B 4A 71 6D 7E 01 37 F8 E0 A3 1E 58 6B D6 D2 C0 26 53 B8 98 B0 09 DF 5A 20 22 E9 A3 DD D0 9E 98 F0 4C A0
11 99 AD C0 48 DA D1 37 3B 2A AF 72 43 91 BD 66 6E 23 56 C7 5A E0 5E 74 33 BF 3A 86 40 07 F3 7D 95 B1 38 70 23 05 30 1B 5E A5 C7 07 13 84 A4 A9 90 B3 66 B5 11 A9 A9 61 72 3C 15 DC EC 94 3E 55 93 83 64 C0 BF 79 2F C3 22 6C B1 C4 AB 5B 12
83 A0 B0 8F 26 98 2F 47 47 7B 2B 10 49 21 80 5F 14 53 0B 4E DA 90 E3 A0 82 98 C8 89 66 63 7A A8 AA A7 EC 08 2D 4B 10 BA 70 A0 3A 93 D1 75 07 59 1C 7D C8 90 55 26 AB B4 90 91 C9 72 10 EB 82 99 3C E8 A3 A7 F0 73 D4 FA 81 9F 99 BF 70 B7 A1
08 BB 58 A8 CC 2D C1 FB BB B7 23 89 CB F0 92 BB A8 65 B3 E4 6C 6B 59 20 88 B5 53 12 EB 49 EB A3 13 B3 78 A5 F1 3A 70 86 8B 8B 96 96 7F 23 9C 37 19 0C C0 3C 0B 14 8F 67 78 4A F5 03 04 04 61 CB E4 AD 37 D0 65 57 26 8B 24 DC 8E 15 A4 AB
CA 6B 3C A4 85 1D 6E 70 74 86 C7 9D B9 8B 4D DD 48 39 6D AB 00 9C 52 17 C3 C3 AD F6 10 A0 DE 0A B5 86 C7 49 5D 94 3A C5 39 95 02 BC 1E A2 73 6A 2D 16 1C 54 31 68 1D A6 C9 B5 C3 A0 12 3B 1C 72 10 60 3E 3B 23 6A 68 92 DF CC 86 91 E7 9E
2B 72 01 7B C4 34 07 61 05 82 53 94 E6 2C CC 30 74 A3 A7 08 4D 91 77 7A 14 5C 4D 4F E2 68 66 94 AD 20 34 22 4F 29 2B 29 37 42 25 0B 53 92 F4 CD A9 19 3A 44 8A 23 98 EC 22 E9 01 24 6D 29 B8 DC 0B B0 45 B9 45 88 43 BA E4 99 3A 32 23 32 6C 66
31 26 FC 06 97 C2 2E 7D 39 B9 AA 17 69 3C 94 66 FA C3 3B C9 E4 5C 75 BC 59 8B 60 14 A7 26 83 FF 72 2C 0D F7 4F E9 88 B0 33 67 6D F3 C2 C0 1A 5B 69 2F 61 48 53 94 9A FC 67 00 03 7A 8C 8E CB 4E 4C 2C 49 E0 82 8C DA B8 AF DA 6A 6D B5 08
9D CA A9 A5 5A E2 91 95 3C 70 5A 3C C0 B0 D2 AD CE 61 A9 8F C4 54 15 8B 3C A9 31 21 D8 28 CD AC 99 09 A2 37 B4 46 53 2F AF E4 AF E8 F2 55 CC FC 4F 4B 78 A4 FC 28 64 10 17 C6 61 71 2F 90 25 26 E0 C5 38 2A C6 4C A4 99 9E F8 D6 3B 1C
48 C0 AE DC B0 B6 1C 85 B3 75 4F B4 5A 14 3F C0 90 CD E6 B7 18 12 6E 34 52 C1 B5 A3 1B E4 17 72 1F F8 25 20 88 21 68 B1 AC 3C E4 A4 2E 53 6C CC 0A 82 32 07 76 C5 23 8C C0 40 07 00 C0 B0 C2 4A 48 D3 B7 13 00 11 C5 88 73 26 F2 EA 96 15 55
CC 3E EB 0F 80 28 93 D4 49 B4 1D 52 18 44 B8 88 6D F5 BC AD B8 60 BF F2 BC 1E 43 34 23 C8 57 4B 65 2F E6 29 97 C8 39 39 BD E7 0F 2D C2 AF 53 DA AA 56 C1 7D 4B 93 54 D9 67 C5 D9 F5 1E 7B 86 49 57 45 85 58 56 84 32 42 B5 39 60 62 40 C2 96
B1 24 9A DA B4 8D FD 8A 9C 5C C0 87 1E 19 12 FD FA 32 DB 69 C3 B3 7B 76 84 15 70 C9 69 73 7A F6 AB D7 A1 21 4C F0 63 06 42 37 7D F3 3A B5 D0 85 EF D6 02 FD F1 6D 4A 3C 55 AB BA 8A 0F 45 A6 5C C8 53 19 F6 A1 75 06 90 A0 26 2D A3 90
9B AB 18 53 2C E9 0F 05 83 3D 0E 49 76 33 D8 4D 8E 72 29 2F FB 51 3B 42 5F 76 C4 AE 54 45 E6 91 8D 2B 45 F0 9A 1E 65 C0 3A D7 99 F5 AE F1 9D E9 45 D3 A3 CD DB 7A 74 E0 3D ED 7F 6D 00 1D 00 20 FA D2 73 2A 7D BB 08 13 72 9B 38 A0 89 F4
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D2 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 C0 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B DD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
dresae (0x3a3a) 00
```

Source: Fiddler Capture

[Thumbprint]
129157F7A5852AD23DF17FE2934E6396DB6B0251

[Signature Algorithm]
sha256RSA(1.2.840.113549.1.1.11)

First decryption
algorithm

[Public Key]

Algorithm: RSA
Length: 2048

Key Blob: 30 82 01 0a 02 82 01 01 00 ca b8 66 5f 8a 37 94 14 e1 2f e3 49 cf d7 ad cb 20 8c 54 93 6b af 66 56 02 11 26 db 20 13 00 a0 ce fe ae e9 ac 8d 68 8f a0 56 fd ff b4 b1 4b 9c 54 ff 55 4a 95 76 b9 db 3c 77 5e 9a b1 dd 30 74 36 77 cb 92 12 df 17 62 6d 2a aa 74 3c f6 68 c8 34 8d eb 69 97 9e f2 43 02 fa d7 76 99 c0 bc b3 56 2d 17 ca 39 c8 00 5d 91 c8 1d 9b c9 70 c8 c1 b0 da 40 23 3e 9a fe a9 ed e4 60 ce 15 b0 af 43 a3 07 a5 81 25 dd c3 92 e7 72 9e 27 df bd 01 05 2a b4 60 b8 83 6b fa bc db 97 51 47 a8 2e d0 52 7f c2 0a 1c 02 96 d2 c7 fd 04 a5 7e 78 04 ee 59 ed 33 9a e4 cc 42 bb 87 36 95 ac 4b 9b 6b 4f 96 4c ed 20 f7 bd 71 4b bc d8 91 81 3c be 56 c8 05 a6 e1 7f 66 85 88 46 91 96 a3 e4 e8 bb 45 f5 b8 78 25 4b 02 cb 23 e6 54 94 ed 94 f6 87 3d 29 55 31 eb 59 e9 a1 54 4c 50 b0 74 2c a7 b6 1e 7f 23 02 03 01 00 01

Parameters: 05 00

[Extensions]

* Key Usage(2.5.29.15):
Digital Signature, Key Encipherment (a0)

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second encryption algorithm
00000019	63 6F 6D 3A 34 34 33 20 48 54 54 50 2F 31 2E 31 0D 0A 48 6F 73 74 3A 20 77									com:443 HTTP/1.1..Host: w
00000032	77 77 2E 68 69 67 67 69 6E 62 6F 74 68 61 6D 2E 63 6F 6D 3A 34 34 33 0D 0A									ww.migginbotham.com:443..
0000004B	43 6F 6E 6E 65 63 74 69 6F 6E 3A 20 6B 65 65 70 2D 61 6C 69 76 65 0D 0A 55									Connection: keep-alive..U
00000064	73 65 72 2D 41 67 65 6E 74 3A 20 4D 6F 7A 69 6C 6C 61 2F 35 2E 30 20 28 57									ser-Agent: Mozilla/5.0 (W
0000007D	69 6E 64 6F 77 73 20 4E 54 20 31 30 2E 30 3B 20 57 69 6E 36 34 3B 20 78 36									indows NT10.0; Win64; x6
00000096	34 29 20 41 70 70 6C 65 57 65 62 4B 69 74 2F 35 33 37 2E 33 36 20 28 4B 48									4) AppleWebKit/537.36 (KH
000000AF	54 4D 4C 2C 20 6C 69 6B 65 20 47 65 63 6B 6F 29 20 43 68 72 6F 6D 65 2F 31									TML, like Gecko) Chrome/1
000000C8	32 36 2E 30 2E 30 2E 30 20 53 61 66 61 72 69 2F 35 33 37 2E 33 36 0D 0A 0D									26.0.0.0 Safari/537.36...
000000E1	0A 41 20 53 53 4C 76 33 2D 63 6F 6D 70 61 74 69 62 6C 65 20 43 6C 69 65 6E									.A SSLv3-compatible Clie
000000FA	74 48 65 6C 6C 6F 20 68 61 6E 64 73 68 61 6B 65 20 77 61 73 20 66 6F 75 6E									tHello handshake was foun
00000113	64 2E 20 46 69 64 64 6C 65 72 20 65 78 74 72 61 63 74 65 64 20 74 68 65 20									d. Fiddler extracted the
0000012C	70 61 72 61 6D 65 74 65 72 73 20 62 65 6C 6F 77 2E 0A 0A 53 65 63 75 72 65									parameters below...Secure
00000145	20 50 72 6F 74 6F 63 6F 6C 3A 20 54 4C 53 20 31 2E 33 0A 43 69 70 68 65 72									Protocol: TLS 1.3.Cipher
0000015E	20 53 75 69 74 65 3A 20 54 4C 53 5F 41 45 53 5F 32 35 36 5F 47 43 4D 5F 53									Suite: TLS_AES_256_GCM_S
00000177	48 41 33 38 34 0A 0A 52 65 63 6F 72 64 20 4C 61 79 65 72 20 56 65 72 73 69									HA384..Record Layer Versi
00000190	6F 6E 3A 20 33 2E 33 20 28 54 4C 53 2F 31 2E 32 29 0A 52 61 6E 64 6F 6D 3A									on: 3.3 (TLS/1.2).Random:
000001A9	20 46 36 20 39 44 20 31 45 20 30 35 20 33 44 20 35 38 20 35 33 20 35 30 20									F6 9D 1E 05 3D 58 53 50
000001C2	43 35 20 33 38 20 43 42 20 36 38 20 45 39 20 42 31 20 37 31 20 42 45 20 30									C5 38 CB 68 E9 B1 71 BE 0
000001DB	32 20 37 37 20 41 37 20 46 41 20 41 42 20 33 46 20 43 43 20 31 44 20 39 37									2 77 A7 FA AB 3F CC 1D 97
000001F4	20 31 42 20 34 43 20 41 46 20 41 42 20 37 41 20 43 44 20 36 39 0A 22 54 69									1B 4C AF AB 7A CD 69."Ti
0000020D	6D 65 22 3A 20 32 31 2D 30 39 2D 31 39 37 32 20 30 38 3A 33 33 3A 31 38 0A									me": 21-09-1972 08:33:18.
00000226	53 65 73 73 69 6F 6E 49 44 3A 20 35 45 20 42 33 20 34 42 20 37 30 20 31 32									SessionID: 5E B3 4B 70 12
0000023F	20 34 44 20 32 43 20 43 42 20 36 41 20 35 42 20 39 41 20 36 33 20 38 39 20									4D 2C CB 6A 5B 9A 63 89

Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream																								
000004C9	36	37	20	39	38	20	39	46	20	38	42	20	38	38	20	44	39	20	41	30	20	43	36	20	31	67	98	9F	8B	88	D9	A0	C6	1
000004E2	35	20	43	43	20	33	37	20	42	42	20	35	31	20	39	31	20	31	44	20	33	30	20	43	45	5	CC	37	BB	51	91	1D	30	CE
000004FB	20	38	44	20	46	36	20	38	45	20	35	46	20	35	34	20	43	32	20	35	41	20	45	33	20	8D	F6	8E	5F	54	C2	5A	E3	
00000514	38	39	20	32	43	20	37	42	20	38	39	20	41	41	20	42	42	20	43	35	20	41	46	20	45	39	2C	7B	89	AA	BB	C5	AF	E
0000052D	39	20	35	36	20	31	30	20	42	38	20	31	35	20	35	37	20	39	34	20	41	31	20	41	31	9	56	10	B8	15	57	94	A1	A1
00000546	20	30	43	20	32	41	20	37	33	20	33	45	20	42	34	20	38	35	20	37	41	20	39	35	20	0C	2A	73	3E	B4	85	7A	95	
0000055F	34	34	20	33	34	20	30	37	20	31	39	20	43	38	20	33	33	20	33	46	20	45	43	20	39	44	34	07	19	C8	33	3F	EC	9
00000578	43	20	34	35	20	34	36	20	30	42	20	38	34	20	43	39	20	31	43	20	32	32	20	45	31	C	45	46	0B	84	C9	1C	22	E1
00000591	20	33	43	20	39	46	20	38	37	20	37	31	20	41	33	20	35	42	20	43	33	20	37	32	20	3C	9F	87	71	A3	5B	C3	72	
000005AA	42	30	20	45	31	20	43	39	20	44	35	20	43	41	20	39	44	20	43	34	20	36	39	20	42	B0	E1	C9	D5	CA	9D	C4	69	B
000005C3	30	20	30	35	20	35	42	20	39	44	20	45	32	20	41	34	20	32	42	20	35	36	20	46	35	0	05	5B	9D	E2	A4	2B	56	F5
000005DC	20	42	46	20	38	32	20	38	34	20	36	37	20	39	43	20	43	36	20	36	35	20	35	46	20	BF	82	84	67	9C	C6	65	5F	
000005F5	38	32	20	33	45	20	42	32	20	46	30	20	42	42	20	46	35	20	33	42	20	41	46	20	30	82	3E	B2	F0	BB	F5	3B	AF	0
0000060E	46	20	38	36	20	31	31	20	41	34	20	31	43	20	30	30	20	35	41	20	34	36	20	36	39	F	86	11	A4	1C	00	5A	46	69
00000627	20	44	39	20	35	37	20	41	46	20	33	31	20	43	36	20	43	32	20	46	34	20	46	30	20	D9	57	AF	31	C6	C2	F4	F0	
00000640	33	38	20	46	37	20	45	43	20	39	37	20	36	41	20	32	31	20	37	38	20	43	34	20	41	38	F7	EC	97	6A	21	78	C4	A
00000659	38	20	41	41	20	42	34	20	38	35	20	43	45	20	43	39	20	38	35	20	36	37	20	44	37	8	AA	B4	85	CE	C9	85	67	D7
00000672	20	37	41	20	30	43	20	45	42	20	41	42	20	37	39	20	31	44	20	38	33	20	33	41	20	7A	0C	EB	AB	79	1D	83	3A	
0000068B	42	32	20	33	42	20	36	30	20	44	36	20	41	33	20	43	32	20	30	31	20	38	37	20	41	B2	3B	60	D6	A3	C2	01	87	A
000006A4	37	20	46	37	20	43	43	20	42	38	20	41	30	20	45	35	20	31	45	20	31	46	20	35	35	7	F7	CC	B8	A0	E5	1E	1F	55
000006BD	20	38	38	20	33	34	20	36	35	20	32	46	20	42	41	20	43	41	20	30	36	20	39	43	20	88	34	65	2F	BA	CA	06	9C	
000006D6	33	42	20	31	37	20	39	44	20	41	36	20	31	31	20	42	32	20	33	39	20	39	46	20	33	3B	17	9D	A6	11	B2	39	9F	3
000006EF	34	20	30	30	20	36	43	20	43	42	20	44	30	20	35	33	20	33	45	20	35	37	20	31	36	4	00	6C	CB	D0	53	3E	57	16

Source: Fiddler Capture

Encrypted HTTPS traffic flows through this CONNECT tunnel. HTTPS Decryption is enabled in Fiddler, so decrypted sessions running in this tunnel will be shown in the Web Sessions list.

Secure Protocol: TLS 1.3

Cipher Suite: TLS_AES_256_GCM_SHA384

== Server Certificate ==

[Version]
V3

[Subject]
CN=higginbotham.com
Simple Name: higginbotham.com
DNS Name: higginbotham.com

[Issuer]
CN=GTS CA 1P5, O=Google Trust Services LLC, C=US

Second decryption
algorithm

Source: Fiddler Capture

The standard defines four record message types, including a handshake message type.

The handshake messages are communicated to establish a secure channel for TLS communication. A client device and a server negotiate an AEAD algorithm for encrypting TLS record message data. It discloses that after Hello handshake messages i.e., a ClientHello message and a ServerHello message, all handshake messages are encrypted with the negotiated encryption algorithm. One of the handshake messages after hello handshake messages i.e., an authentication message from the client, comprises a digital certificate encrypted by a signature encryption algorithm and a certificate verify message comprising information related to the signature decryption algorithm.

As shown below, the digital certificate is encrypted with the signature encryption algorithm and the certificate verify message, associated with the encrypted digital certificate, has a signature algorithm extension field that provides information related to the signature decryption algorithm. The authentication message is a TLS plaintext handshake message. This message is again encrypted with the negotiated AEAD encryption algorithm, e.g., recursive security protocol. The AEAD encrypted message is communicated between the client and the server.

Further, the AEAD encrypted message comprises a ciphertext (e.g., encrypted ciphertext after the encryption by the second encryption algorithm), nonce (e.g., associating second decryption algo), key and associated data. The maximum length of nonce is a cipher suit specific element. The nonce and associated data are utilized in decryption of the AEAD encrypted message.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is verified, decrypted, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher-level protocols to be multiplexed over the same record layer. This document specifies four content types: handshake, application_data, alert, and change_cipher_spec. The change_cipher_spec record is used only for compatibility purposes (see [Appendix D.4](#)).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2. Protocol Overview

Negotiating encryption algorithm

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

<https://datatracker.ietf.org/doc/html/rfc8446>

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering, an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) that uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

Negotiating encryption algos

<https://datatracker.ietf.org/doc/html/rfc8446>

5.1. Record Layer

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^{14} bytes or less. Message boundaries are handled differently depending on the underlying ContentType. Any future content types MUST specify appropriate rules. Note that these rules are stricter than what was enforced in TLS 1.2.

Handshake messages MAY be coalesced into a single TLSPlaintext record or fragmented across several records, provided that:

- Handshake messages MUST NOT be interleaved with other record types. That is, if a handshake message is split over two or more records, there MUST NOT be any other records between them.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

5.2. Record Payload Protection

The record protection functions translate a TLSPlaintext structure into a TLSCiphertext structure. The deprotection functions reverse the process. In TLS 1.3, as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Associated Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in Section 2.1 of [RFC5116]. The key is either the client_write_key or the server_write_key, the nonce is derived from the sequence number and the client_write_iv or server_write_iv (see [Section 5.3](#)), and the additional data input is the record header.

I.e.,

```
additional_data = TLSCiphertext.opaque_type ||  
                  TLSCiphertext.legacy_record_version ||  
                  TLSCiphertext.length
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The AEAD approach enables applications that need cryptographic security services to more easily adopt those services. It benefits the application designer by allowing them to focus on important issues such as security services, canonicalization, and data marshaling, and relieving them of the need to design crypto mechanisms that meet their security goals. Importantly, the security of an AEAD algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AEAD of the need to consider security aspects such as the relative order of authentication and encryption and the security of the particular combination of cipher and MAC, such as the potential loss of confidentiality through the MAC. The application designer that uses the AEAD interface need not select a particular AEAD algorithm during the design stage. Additionally, the interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.1. Authenticated Encryption

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

2.2. Authenticated Decryption

Second decryption algorithm

The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [RFC5116]. The length of the TLS per-record nonce (iv_length) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm (see [RFC5116], Section 4). An AEAD algorithm where N_MAX is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

All handshake messages after the ServerHello are now encrypted. The newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4. Authentication Messages

As discussed in [Section 2](#), TLS generally uses a common set of messages for authentication, key confirmation, and handshake integrity: Certificate, CertificateVerify, and Finished. (The PSK binders also perform key confirmation, in a similar fashion.) These three messages are always sent as the last messages in their handshake flight. The Certificate and CertificateVerify messages are only sent under certain circumstances, as defined below. The Finished message is always sent as part of the Authentication Block.

These messages are encrypted under keys derived from the [\[sender\]](#)_handshake_traffic_secret.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

Figure 1 below shows the basic full TLS handshake:

```

Client
Key ^ ClientHello
Exch | + key_share*
    | + signature_algorithms*
    | + psk_key_exchange_modes*
    v + pre_shared_key* ----->

Server
ServerHello ^ Key
            + key_share* | Exch
            + pre_shared_key* v
            {EncryptedExtensions} ^ Server
            {CertificateRequest*} v Params
            {Certificate*} ^
            {CertificateVerify*} | Auth
            {Finished} v
<----- [Application Data*]

Auth ^ {Certificate*}
    | {CertificateVerify*}
    v {Finished}
    [Application Data]
<-----> [Application Data]

```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2. Certificate

This message conveys the endpoint's certificate chain to the peer.

The server MUST send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client MUST send a Certificate message if and only if the server has requested client authentication via a CertificateRequest message (Section 4.3.2). If the server requests client authentication but no suitable certificate is available, the client MUST send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0). A Finished message MUST be sent regardless of whether the Certificate message is empty.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.2.3. Client Certificate Selection

The following rules apply to certificates sent by the client:

- The certificate type MUST be X.509v3 [[RFC5280](#)], unless explicitly negotiated otherwise (e.g., [[RFC7250](#)]).
- If the "certificate_authorities" extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in [Section 4.3.2](#). Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client, as described in [Section 4.2.5](#).

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.4.3. Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

Structure of this message:

```
struct {  
    SignatureScheme algorithm;  
    opaque signature<0..2^16-1>;  
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see Section 4.2.3 for the definition of this type). The signature is a digital signature using that algorithm. The content that is covered under the signature is the hash output as described in Section 4.4.1, namely:

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

4.3.2. Certificate Request

A server which is authenticating with a certificate MAY optionally request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

Structure of this message:

```
struct {  
    opaque certificate_request_context<0..2^8-1>;  
    Extension extensions<2..2^16-1>;  
} CertificateRequest;
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

certificate_request_context: An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The certificate_request_context MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). This field SHALL be zero length unless used for the post-handshake authentication exchanges described in [Section 4.6.2](#). When requesting post-handshake authentication, the server SHOULD make the context unpredictable to the client (e.g., by randomly generating it) in order to prevent an attacker who has temporary access to the client's private key from pre-computing valid CertificateVerify messages.

extensions: A set of extensions describing the parameters of the certificate being requested. The "signature_algorithms" extension MUST be specified, and other extensions may optionally be included if defined for this message. Clients MUST ignore unrecognized extensions.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4.3.2>

- RSASSA-PSS signature schemes are defined in [Section 4.2.3](#).
- The "supported_versions" ClientHello extension can be used to negotiate the version of TLS to use, in preference to the legacy_version field of the ClientHello.
- The "signature_algorithms_cert" extension allows a client to indicate which signature algorithms it can validate in X.509 certificates.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

If sent by a client, the signature algorithm used in the signature MUST be one of those present in the supported_signature_algorithms field of the "signature_algorithms" extension in the CertificateRequest message.

In addition, the signature algorithm MUST be compatible with the key in the sender's end-entity certificate. RSA signatures MUST use an RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". The SHA-1 algorithm MUST NOT be used in any signatures of CertificateVerify messages.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

4.2.3. Signature Algorithms

TLS 1.3 provides two extensions for indicating which signature algorithms may be used in digital signatures. The "signature_algorithms_cert" extension applies to signatures in certificates, and the "signature_algorithms" extension, which originally appeared in TLS 1.2, applies to signatures in CertificateVerify messages. The keys found in certificates MUST also be of appropriate type for the signature algorithms they are used with. This is a particular issue for RSA keys and PSS signatures, as described below. If no "signature_algorithms_cert" extension is present, then the "signature_algorithms" extension also applies to signatures appearing in certificates. Clients which desire the server to authenticate itself via a certificate MUST send the "signature_algorithms" extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension, then the server MUST abort the handshake with a "missing_extension" alert (see [Section 9.2](#)).

The "signature_algorithms_cert" extension was added to allow implementations which supported different sets of algorithms for certificates and in TLS itself to clearly signal their capabilities. TLS 1.2 implementations SHOULD also process this extension. Implementations which have the same policy in both cases MAY omit the "signature_algorithms_cert" extension.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

As shown below, the receiving party will be able to decrypt the encrypted message with the provided signature decryption algorithm information i.e., SHA-256 RSA decryption algorithm.

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

	<p>The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A <i>cryptographic hash function</i> is a function that computes a <i>message authentication code</i> from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.</p> <p>https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf</p>
39. The software system or computer program of claim 38, wherein the decrypting is done using a key associated with each decryption algorithm.	The standard practices the method such that the decrypting is done using a key (e.g., decryption key) associated with each decryption algorithm (e.g., signature decryption algorithm such as SHA-256RSA, etc., and AEAD decryption algorithm such as TLS_AES_256_GCM_SHA384, etc.).

The Higginbotham Difference

About Us

Careers

Blog

Locations

Become a Higginbotham Partner

Report a Claim

H Higginbotham™

Business Insurance

Employee Benefits

HR Services

Financial Services

Personal Insurance

Captive Insurance

Let's Talk

Insurance Brokers Who are Principled, Perceptive and Personal

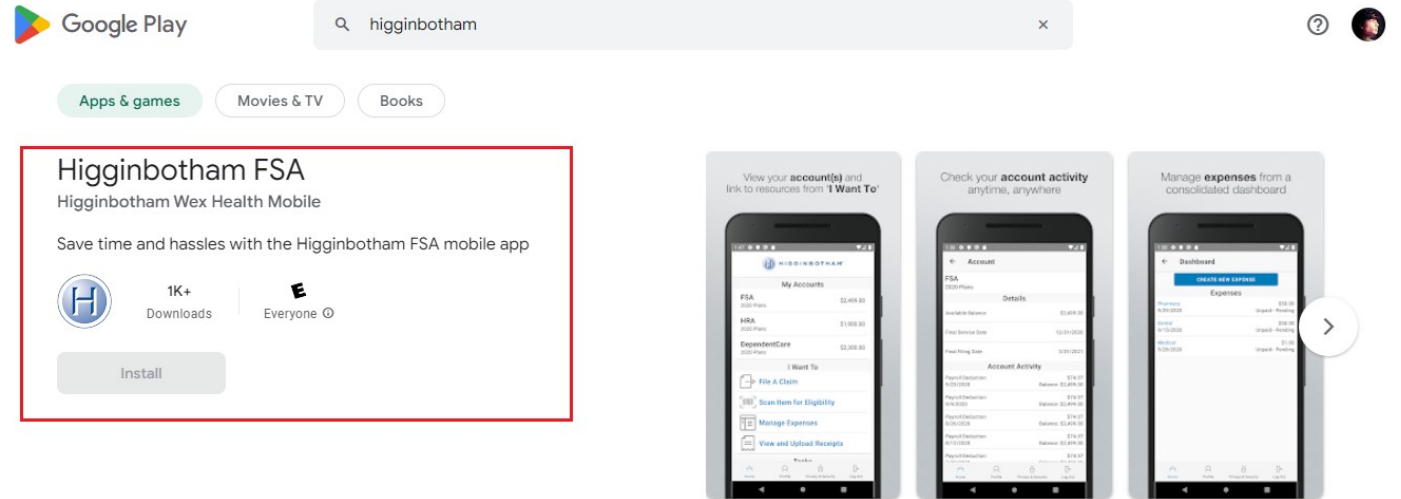
A single source solution inspired by your needs.

Request a Quote →

Let's Talk →

PAUSE VIDEO

<https://www.higginbotham.com/>



<https://play.google.com/store/search?q=higginbotham&c=apps&hl=en&gl=US>

```
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 DD 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
crease (0x3a3a) 00
```

Source: Fiddler Capture

As shown below, the signature decryption algorithm utilizes a private key for a first decryption and the AEAD decryption algorithm uses a key K. Both the decryption techniques are decrypting using their respective associated keys.

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p><u>2.2. Authenticated Decryption</u></p> <p>The <u>authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).</u></p> <p>https://datatracker.ietf.org/doc/html/rfc5116</p> <p>The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. <u>The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.</u></p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
40. The software system or computer program of claim 39, wherein the key is resident in hardware of the target unit or the key is retrieved from a	The standard utilized by the accused instrumentality practices the method such that the key is resident in hardware (e.g., stored in a memory storage of the server such as a database, RAM, etc.) of the target unit (e.g., server of the accused instrumentality) or the key is retrieved from a server.

server.

The screenshot shows the homepage of the Higginbotham website. The header is dark blue with white text for navigation links: "The Higginbotham Difference", "About Us", "Careers", "Blog", "Locations", "Become a Higginbotham Partner", a search icon, and "Report a Claim". Below this is a secondary navigation bar with the Higginbotham logo (a stylized 'H' in a circle) and links for "Business Insurance", "Employee Benefits", "HR Services", "Financial Services", "Personal Insurance", "Captive Insurance", and a red "Let's Talk" button. The main content area features a video player with a woman in a purple top. Overlaid on the video is a white box with the text "Insurance Brokers Who are Principled, Perceptive and Personal" and "A single source solution inspired by your needs." Below this text are two buttons: "Request a Quote →" and "Let's Talk →". A "PAUSE VIDEO" control is visible in the bottom right corner of the video player. At the bottom of the screenshot is the URL <https://www.higginbotham.com/>.

server.

The Higginbotham Difference About Us Careers Blog Locations Become a Higginbotham Partner Search Report a Claim

Higginbotham™ Business Insurance Employee Benefits HR Services Financial Services Personal Insurance Captive Insurance Let's Talk


Insurance Brokers Who are Principled, Perceptive and Personal

A single source solution inspired by your needs.

Request a Quote → Let's Talk →

PAUSE VIDEO

<https://www.higginbotham.com/>



Apps & games


Movies & TV

Books

Higginbotham FSA

Higginbotham Wex Health Mobile

Save time and hassles with the Higginbotham FSA mobile app



1K+


Downloads

Everyone


Age

Install

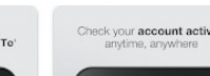
View your **account(s)** and link to resources from **"I Want To"**



Check your **account activity** anytime, anywhere



Manage **expenses** from a consolidated dashboard



<https://play.google.com/store/search?q=higginbotham&c=apps&hl=en&gl=US>

```

1A 36 23 01 54 D1 F6 FB F9 09 32 32 D7 CC C6 34 38 24
    extended_master_secret  empty
    ec_point_formats  uncompressed [0x0]
    status_request  OCSP - Implicit Responder
    psk_key_exchange_modes  01 01
    renegotiation_info  00
    0xf0fed
    00 00 01 00 01 8B 00 20 08 F0 C0 F2 E C 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E1 F7 BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 D0 29 05 15
F6 5E AC B2 14 57 6C 0B F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B BD F9 05 0F D4 63 E1 FC 38 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
    server_name  www.higginbotham.com
    0reason (0x3a3a)  00

```

Source: Fiddler Capture



Tech Accelerator

Server hardware guide: Architecture, products and management

3. Random access memory



RAM is the main type of memory in a computing system.

RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as

data to be moved to a storage device. Thus, RAM works very

closely with the processor and must match the processor's

incredible speed and performance. This kind of fast memory

is usually termed dynamic RAM, and several DRAM

variations are available for servers.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>



Tech Accelerator

Server hardware guide: Architecture, products and management

f

X

in



4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>

As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. This is necessary for the ClientHello storage mechanism described in Section 8.2 because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.

<https://datatracker.ietf.org/doc/html/rfc8446#>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

2.2. Authenticated Decryption

The authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).

<https://datatracker.ietf.org/doc/html/rfc5116>

The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

41. The software system or computer program of claim 40, wherein the key is contained in a key data structure.

The standard utilized by the accused instrumentality practices the method such that the key (e.g., private key, Key K, etc.) is contained in a key data structure (e.g., data structure).

The Higginbotham Difference

About Us

Careers

Blog

Locations

Become a Higginbotham Partner

Report a Claim

H Higginbotham™

Business Insurance

Employee Benefits

HR Services

Financial Services

Personal Insurance

Captive Insurance

Let's Talk

Insurance Brokers Who are Principled, Perceptive and Personal

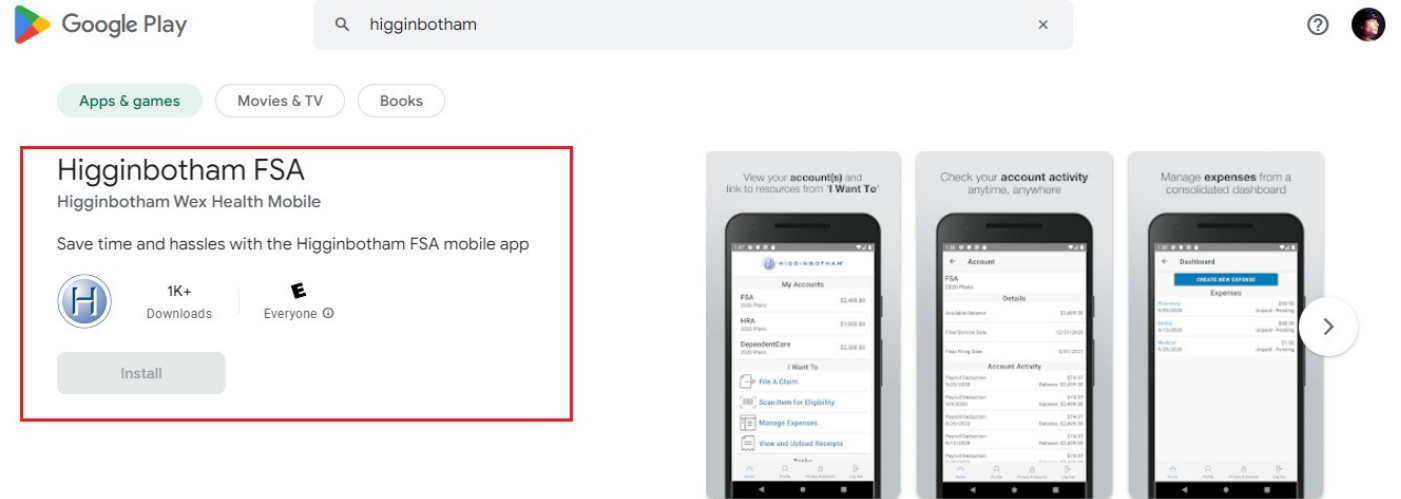
A single source solution inspired by your needs.

Request a Quote →

Let's Talk →

PAUSE VIDEO

<https://www.higginbotham.com/>



<https://play.google.com/store/search?q=higginbotham&c=apps&hl=en&gl=US>

```
7A 36 23 01 54 D1 F6 FB F9 09 92 32 D7 CC C6 34 38 24
extended_master_secret empty
ec_point_formats uncompressed [0x0]
status_request OCSP - Implicit Responder
psk_key_exchange_modes 01 01
renegotiation_info 00
0xfe0d 00 00 01 00 01 8B 00 20 08 8F C0 F2 2E C4 97 D9 A5 41 74 90 DC B9 00 DF 96 AC CE 3B D4 B6 C4 68 48 15 D0 44 C5 08 4A 5E 00 D0 A7 B0 A8 9E C4 E4 1F BB 61 CD 40 13 BB 50 B2 C5 69 CC 12 85 91 94 13 32 DD 29 05 15
F6 5E AC B2 14 57 6C 00 FB F3 90 2C 7E 8F 33 09 60 EB EC 99 5C CD 15 71 45 CC CF 2D 43 0A AF 55 84 39 DF AF 78 19 F7 A8 BC B3 30 C6 42 13 6B B6 B6 52 7A 00 F6 85 56 93 A2 DF BD 8D C7 84 26 48 A3 7A 84 49 D9 2A 13 70 C7 DC 03 CD 58 6E
2D 8A 8B BD F9 05 0F D4 63 E1 FC 35 82 4F 90 98 5C AB B9 04 52 E5 A7 B5 58 20 C6 03 9F 26 D4 7A 32 7B 1D DB BC 97 E1 43 E7 3E 21 46 61 79 37 45 BF 90 B6 FB 61 AE 44 A1 D9 62 79 85 26 3D 37 D4 40 BE 29 E1 61 60 94 0C 47 88 C9 21 21 E8 8C
E4 C8 4A D5 F3 3C D6 8F 9A DF C3 0A 9F 27 E4 C6 05 53 5F 83 38 B9 5A
server_name www.higginbotham.com
orease(0x3a3a) 00
```

Source: Fiddler Capture

The accused instrumentality utilizes a server to establish a secure TLS communication with a client. The server must comprise a memory storage and store data according to a data structure to implement the standard efficiently.



Tech Accelerator

Server hardware guide: Architecture, products and management

3. Random access memory



RAM is the main type of memory in a computing system.

RAM holds the software instructions and data needed by the processor, along with any output from the processor, such as data to be moved to a storage device. Thus, RAM works very closely with the processor and must match the processor's incredible speed and performance. This kind of fast memory is usually termed dynamic RAM, and several DRAM variations are available for servers.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>



Tech Accelerator

Server hardware guide: Architecture, products and management



4. Hard disk drive

This hardware is responsible for reading, writing and positioning of the hard disk, which is one technology for data storage on server hardware. Developed at IBM in 1953, the hard disk drive (HDD) has evolved over time from the size of a refrigerator to the standard 2.5-inch and 3.5-inch form factors.

<https://www.techtarget.com/searchdatacenter/feature/Drill-down-to-basics-with-these-server-hardware-terms>

A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it.

In computer science and computer programming, a data structure may be selected or designed to store data for the purpose of using it with various algorithms. In some cases, the algorithm's basic operations are tightly coupled to the data structure's design. Each data structure contains information about the data values, relationships between the data and -- in some cases -- functions that can be applied to the data.

<https://www.techtarget.com/searchdatamanagement/definition/data-structure>

As shown below, the server comprises a memory storage to store messages for establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. Both the decryption techniques are decrypting using their respective associated keys. A server must have a storage to store information pertaining to these algorithms and their corresponding keys such as private key, Key K, etc., to establish secure TLS communication with a client.

	<p>Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. <u>This is necessary for the ClientHello storage mechanism described in Section 8.2</u> because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.</p> <p>https://datatracker.ietf.org/doc/html/rfc8446#</p>
--	--

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q , compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de \equiv 1 \pmod{\varphi(n)}$. We know that $de - 1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

First encryption

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

First decryption

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

	<p><u>2.2. Authenticated Decryption</u></p> <p>The <u>authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).</u></p> <p>https://datatracker.ietf.org/doc/html/rfc5116</p> <p>The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. <u>The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.</u></p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>47. The software system or computer program of claim 39, wherein each encryption algorithm is a symmetric key system or an</p>	<p>The standard practices the method such that each encryption algorithm (e.g., signature encryption algorithm i.e., SHA256RSA, etc., and AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.) is a symmetric key system (e.g., AEAD encryption algorithm, etc.) or an asymmetric key system (e.g., signature encryption algorithm).</p> <p>As shown below, the server comprises a memory storage to store messages for</p>

asymmetric system.	key	<p>establishing secure TLS communication. the standard discloses multiple signature encryption algorithms for a first encryption and multiple AEAD encryption algorithms for the second encryption. A signature decryption algorithm utilizes a private key for decrypting the first bitstream encrypted with the signature encryption and an AEAD decryption algorithm uses a key K for decrypting the second bitstream encrypted with the AEAD encryption. The standard defines the signature encryption algorithm as an asymmetric cryptography algorithm and the AEAD encryption algorithm as the symmetric cryptography algorithm.</p> <p>Because the ClientHello indicates the time at which the client sent it, it is possible to efficiently determine whether a ClientHello was likely sent reasonably recently and only accept 0-RTT for such a ClientHello, otherwise falling back to a 1-RTT handshake. <u>This is necessary for the ClientHello storage mechanism described in Section 8.2</u> because otherwise the server needs to store an unlimited number of ClientHellos, and is a useful optimization for self-contained single-use tickets because it allows efficient rejection of ClientHellos which cannot be used for 0-RTT.</p> <p>https://datatracker.ietf.org/doc/html/rfc8446#</p> <p>Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via <u>asymmetric cryptography (e.g., RSA [RSA], the Elliptic Curve Digital Signature Algorithm (ECDSA) [ECDSA], or the Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032])</u> or a symmetric pre-shared key (PSK).</p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-4</p>
--------------------	-----	---

cipher_suites: A list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. Values are defined in [Appendix B.4](#). If the list contains cipher suites that the server does not recognize, support, or wish to use, the server MUST ignore those cipher suites and process the remaining ones as usual. If the client is attempting a PSK key establishment, it SHOULD advertise at least one cipher suite indicating a Hash associated with the PSK.

<https://datatracker.ietf.org/doc/html/rfc8446#section-4>

The "extension_data" field of these extensions contains a SignatureSchemeList value:

```
enum {  
    /* RSASSA-PKCS1-v1_5 algorithms */  
    rsa_pkcs1_sha256(0x0401),  
    rsa_pkcs1_sha384(0x0501),  
    rsa_pkcs1_sha512(0x0601),  
  
    /* ECDSA algorithms */  
    ecdsa_secp256r1_sha256(0x0403),  
    ecdsa_secp384r1_sha384(0x0503),  
    ecdsa_secp521r1_sha512(0x0603),  
  
    /* RSASSA-PSS algorithms with public key OID rsaEncryption */  
    rsa_pss_rsae_sha256(0x0804),  
    rsa_pss_rsae_sha384(0x0805),  
    rsa_pss_rsae_sha512(0x0806),  
  
    /* EdDSA algorithms */  
    ed25519(0x0807),  
    ed448(0x0808),  
  
    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */  
    rsa_pss_pss_sha256(0x0809),  
    rsa_pss_pss_sha384(0x080a),  
    rsa_pss_pss_sha512(0x080b),
```

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

There is also a decryption function D that takes a ciphertext and a decryption key K_D , and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B 's encryption key, encrypt a message with it, and send the result to person B . Only someone with B 's decryption key, namely only B , can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

This specification defines the following cipher suites for use with TLS 1.3.

Description	Value
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

<https://datatracker.ietf.org/doc/html/rfc8446#section-1>

The authenticated encryption operation has four inputs, each of which is an octet string:

A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.

A nonce N. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero-length. Applications that can generate distinct nonces SHOULD use the nonce formation method defined in [Section 3.2](#), and MAY use any other method that meets the uniqueness requirement. Other applications SHOULD use zero-length nonces.

A plaintext P, which contains the data to be encrypted and authenticated.

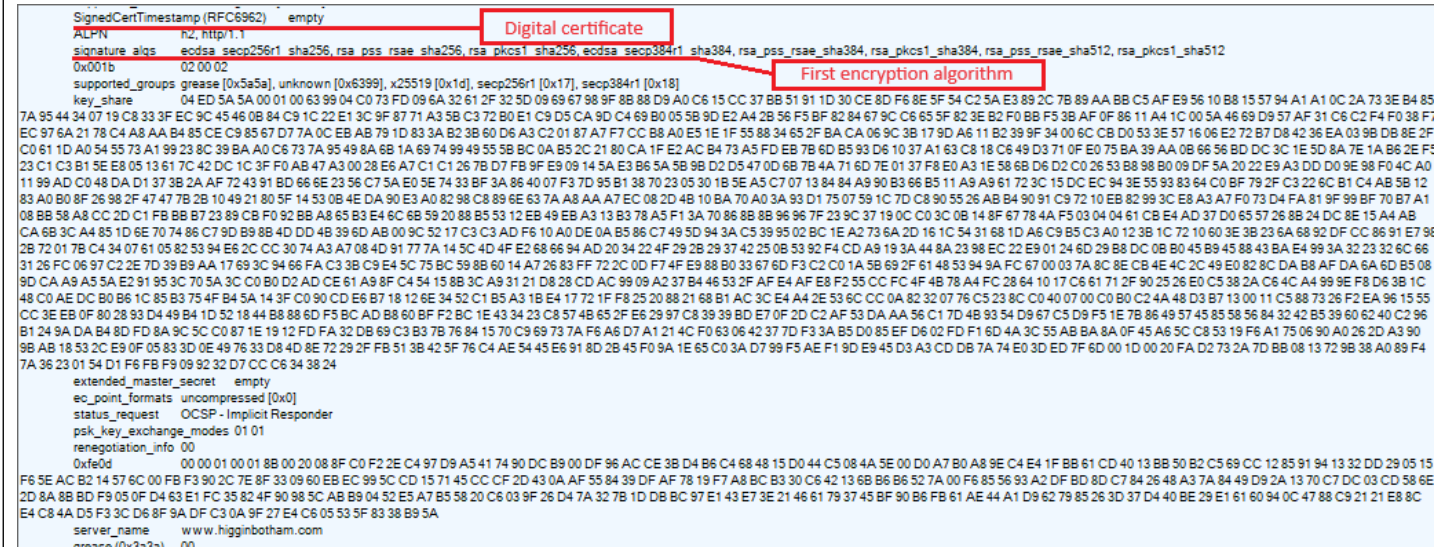
The associated data A, which contains the data to be authenticated, but not encrypted.

<https://datatracker.ietf.org/doc/html/rfc5116>

	<p><u>2.2. Authenticated Decryption</u></p> <p>The <u>authenticated decryption operation has four inputs: K, N, A, and C, as defined above. It has only a single output, either a plaintext value P or a special symbol FAIL that indicates that the inputs are not authentic. A ciphertext C, a nonce N, and associated data A are authentic for key K when C is generated by the encrypt operation with inputs K, N, P, and A, for some values of N, P, and A. The authenticated decrypt operation will, with high probability, return FAIL whenever the inputs N, P, and A were crafted by a nonce-respecting adversary that does not know the secret key (assuming that the AEAD algorithm is secure).</u></p> <p>https://datatracker.ietf.org/doc/html/rfc5116</p> <p>The AEAD output consists of the ciphertext output from the AEAD encryption operation. The length of the plaintext is greater than the corresponding TLSPlaintext.length due to the inclusion of TLSInnerPlaintext.type and any padding supplied by the sender. <u>The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD algorithm.</u></p> <p>https://datatracker.ietf.org/doc/html/rfc8446#section-1</p>
<p>48. The software system or computer program of claim 39, further translatable for associating a first Message Authentication Code</p>	<p>The standard practices associating a first Message Authentication Code (MAC) (e.g., message authentication code with hashing function) or first digital signature with each encrypted bit stream (e.g., encrypted bit stream with the signature encryption algorithm i.e., SHA256RSA, etc., and encrypted bitstream with the AEAD encryption algorithm i.e., TLS_AES_256_GCM_SHA384, etc.).</p> <p>As shown below, the standard discloses a hashing function with each of the encryption</p>

(MAC) or first digital signature with each encrypted bit stream.

algorithm. It performs a message authentication code with the utilized hashing function.



Source: Fiddler Capture

First encryption algorithm

Second encryption algorithm

```
Protocol: TLS 1.3.Cipher
Suite: TLS_AES_256_GCM_S
HA384..Record Layer Versi
on: 3.3 (TLS/1.2)..Random:
F6 9D 1E 05 3D 58 53 50
C5 38 CB 68 E9 B1 71 BE 0
2 77 A7 FA AB 3F CC 1D 97
1B 4C AF AB 7A CD 69."Ti
me": 21-09-1972 08:33:18.
SessionID: 5E B3 4B 70 12
4D 2C CB 6A 5B 9A 63 89
```


Source: Fiddler Capture

Headers	TextView	SyntaxView	WebForms	HexView	Auth	Cookies	Raw	JSON	XML	Second bitstream
000004C9	36 37 20 39 38 20 39 46 20 38 42 20 38 38 20 44 39 20 41 30 20 43 36 20 31									67 98 9F 8B 88 D9 A0 C6 1
000004E2	35 20 43 43 20 33 37 20 42 42 20 35 31 20 39 31 20 31 44 20 33 30 20 43 45									5 CC 37 BB 51 91 1D 30 CE
000004FB	20 38 44 20 46 36 20 38 45 20 35 46 20 35 34 20 43 32 20 35 41 20 45 33 20									8D F6 8E 5F 54 C2 5A E3
00000514	38 39 20 32 43 20 37 42 20 38 39 20 41 41 20 42 42 20 43 35 20 41 46 20 45									39 2C 7B 89 AA BB C5 AF E
0000052D	39 20 35 36 20 31 30 20 42 38 20 31 35 20 35 37 20 39 34 20 41 31 20 41 31									9 56 10 B8 15 57 94 A1 A1
00000546	20 30 43 20 32 41 20 37 33 20 33 45 20 42 34 20 38 35 20 37 41 20 39 35 20									0C 2A 73 3E B4 85 7A 95
0000055F	34 34 20 33 34 20 30 37 20 31 39 20 43 38 20 33 33 20 33 46 20 45 43 20 39									44 34 07 19 C8 33 3F EC 9
00000578	43 20 34 35 20 34 36 20 30 42 20 38 34 20 43 39 20 31 43 20 32 32 20 45 31									C 45 46 0B 84 C9 1C 22 E1
00000591	20 33 43 20 39 46 20 38 37 20 37 31 20 41 33 20 35 42 20 43 33 20 37 32 20									3C 9F 87 71 A3 5B C3 72
000005AA	42 30 20 45 31 20 43 39 20 44 35 20 43 41 20 39 44 20 43 34 20 36 39 20 42									B0 E1 C9 D5 CA 9D C4 69 B
000005C3	30 20 30 35 20 35 42 20 39 44 20 45 32 20 41 34 20 32 42 20 35 36 20 46 35									0 05 5B 9D E2 A4 2B 56 F5
000005DC	20 42 46 20 38 32 20 38 34 20 36 37 20 39 43 20 43 36 20 36 35 20 35 46 20									BF 82 84 67 9C C6 65 5F
000005F5	38 32 20 33 45 20 42 32 20 46 30 20 42 42 20 46 35 20 33 42 20 41 46 20 30									82 3E B2 F0 BB F5 3B AF 0
0000060E	46 20 38 36 20 31 31 20 41 34 20 31 43 20 30 30 20 35 41 20 34 36 20 36 39									F 86 11 A4 1C 00 5A 46 69
00000627	20 44 39 20 35 37 20 41 46 20 33 31 20 43 36 20 43 32 20 46 34 20 46 30 20									D9 57 AF 31 C6 C2 F4 F0
00000640	33 38 20 46 37 20 45 43 20 39 37 20 36 41 20 32 31 20 37 38 20 43 34 20 41									38 F7 EC 97 6A 21 78 C4 A
00000659	38 20 41 41 20 42 34 20 38 35 20 43 45 20 43 39 20 38 35 20 36 37 20 44 37									8 AA B4 85 CE C9 85 67 D7
00000672	20 37 41 20 30 43 20 45 42 20 41 42 20 37 39 20 31 44 20 38 33 20 33 41 20									7A 0C EB AB 79 1D 83 3A
0000068B	42 32 20 33 42 20 36 30 20 44 36 20 41 33 20 43 32 20 30 31 20 38 37 20 41									B2 3B 60 D6 A3 C2 01 87 A
000006A4	37 20 46 37 20 43 43 20 42 38 20 41 30 20 45 35 20 31 45 20 31 46 20 35 35									7 F7 CC B8 A0 E5 1E 1F 55
000006BD	20 38 38 20 33 34 20 36 35 20 32 46 20 42 41 20 43 41 20 30 36 20 39 43 20									88 34 65 2F BA CA 06 9C
000006D6	33 42 20 31 37 20 39 44 20 41 36 20 31 31 20 42 32 20 33 39 20 39 46 20 33									3B 17 9D A6 11 B2 39 9F 3
000006EF	34 20 30 30 20 36 43 20 43 42 20 44 30 20 35 33 20 33 45 20 35 37 20 31 36									4 00 6C CB D0 53 3E 57 16

Source: Fiddler Capture

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 or 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions MD5, SHA-1, and SHA-256. Suppose that H is a cryptographic hash function. To sign a message m , party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B . Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

<https://cs.pomona.edu/~dkauchak/classes/s17/cs52-s17/handouts/encryption.pdf>

The list of supported symmetric encryption algorithms has been pruned of all algorithms that are considered legacy. Those that remain are all Authenticated Encryption with Associated Data (AEAD) algorithms. The cipher suite concept has been changed to separate the authentication and key exchange mechanisms from the record protection algorithm (including secret key length) and a hash to be used with both the key derivation function and handshake message authentication code (MAC).

<https://datatracker.ietf.org/doc/html/rfc8446#section-4>